DATA- AND COMMUNICATION-CENTRIC APPROACHES TO MODEL AND DESIGN FLEXIBLE DEEP NEURAL NETWORK ACCELERATORS

A Dissertation
Presented to
The Academic Faculty

By

Hyoukjun Kwon

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

August 2020

DATA- AND COMMUNICATION-CENTRIC APPROACHES TO MODEL AND DESIGN FLEXIBLE DEEP NEURAL NETWORK ACCELERATORS

Thesis committee:

Dr. Tushar Kirshna, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Michael Pellauer
Architecture Research Group
NVIDIA

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Alexey Tumanov
School of Computer Science
Georgia Institute of Technology

Date approved: July 16, 2020

Whether you think you can or you think you can't, you're right.

Henry Ford

For my parents and brother who sent endless support for my PhD journey

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Tushar Krishna for the guidance and support during my PhD journey in the past five years. Not only knowledge and technical skills, I could also learn many important qualities to be a good researcher from him: commitment, passion, communications skills, and so on. I sincerely appreciate him for being a great role model I could learn from.

I would like honor Dr. Michael Pellauer as my co-advisor in appreciation of mentoring and advising not only during my internship at NVIDIA but also during my PhD journey. I could learn many technical insights and crucial skills for my PhD and research from him, and this thesis could not be completed without those. I would like to thank him for such invaluable assets I will rely on for my future career.

I would also like to thank my thesis committee members, Professor Vivek Sarkar, Professor Hyesoon Kim, and Professor Alexey Tumanov for their encouragement and insightful comments.

I also had the privilege of learning from Dr. Angshuman Parashar, Dr. Joel Emer, Dr. Aamer Jaleel, Dr. Christopher Fletcher, Dr. Neal Crago, and Dr. Steve Keckler at NVIDIA during my internship in Architecture Research Group. I appreciate for invaluable learning opportunities at NVIDA and countless hours for discussing research ideas, which led to major contributions of this thesis.

I was also fortunate to work with Dr. Liangzhen Lai, Dr. Vikas Chandra, Dr. Meng Li, Dr. Pierce Chuang, Dr. Simon Hollis, Dr. Ganesh Venkatesh, and Dr. Yilei Li, during my internship at Facebook and later on. I would like thank them for extremely helpful feedback, comments, and support for my research during my internship, which were from expertise from entire computation stack.

I also thank other collaborators, Ananda Samajdar, Zhongyuan Zhao, Rebert Guirado, Prasanth Chatarasi, Dr. Sergi Abadal, who enabled deeper and inter-disciplinary research.

My PhD journey was motivated by my undergraduate advisor Professor Jihong Kim at SNU and Professor Arvind at MIT. I appreciate their direct and indirect encouragement toward research career.

Finally, thanks to all current Synergy lab members, Mayank Parasar, Ananda Samajdar, Eric Qin, Sheng-Chun (Felix) Kao, Yehowshua Immanuel, Saeed Rashidi, Matthew Denton, Geonhwa Jeong, and William (Jonghoon) Won. And also, thanks to all past Synergy lab members, Vineet Nadella, Fei Wu, Parth Mannan, Srikant Bharadwaj, Brian Lebiednik, and Aniruddh Ramrakhyani.

Lastly, I would like to thank my parents and family for their unconditional and endless support. The journey of PhD was only possible because of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Deep neural network (DNN) enabled high operational performance (i.e., accuracy or quality of outputs) in many applications such as image classification, face recognition, natural language processing, and so on. Since recent DNNs involve billions of multiply-and-accumulate (MAC) operations with millions of parameters, DNN accelerators, specialized hardware for DNN computation, have emerged. However, designing dedicated hardware for each DNN model requires high development costs while DNN models and algorithms rapidly evolve. In addition, specializing a DNN accelerator for one DNN model with limited support for compiler mappings often leads to inefficiency for other DNN models. Therefore, this thesis explores flexible DNN accelerator designs that support diverse compiler mappings (i.e., dataflow + tile sizes for each data dimension) to adapt to new DNN models without re-designing hardware.

This thesis first focuses on the modeling costs and benefits of mapping choices to quantify the potential costs and benefits of mapping choices considering underlying hardware. We codify the cost model and implement *MAESTRO*, and perform case studies that show no single mapping is ideal for all the layers. For the flexible DNN accelerator designs, this thesis addresses the challenge from two perspectives: reconfigurability and heterogeneity.

For the reconfigurability approach, this thesis focuses on the data movement since the cost of data movement dominates in DNN accelerators, and the rearranging the data movement is effectively equivalent to programming a DNN accelerator considering the nature of predefined target application. We propose a light-weight network-on-chip (NoC) architecture, *Microswitch NoC*, specialized for DNN accelerator traffic while providing sufficient flexibility for any dataflow. We also present a reconfigurable DNN accelerator design, *MAERI*, that employs reconfigurable data distribution and reduction NoCs that support all the communication patterns in DNN accelerators and perform reduction inside NoC switches (i.e., in-network-processing style). MAERI enables to map computations on

compute units without underutilizing PEs for any irregular DNN computations resulting from diverse layers and various optimizations (e.g., cross layer mapping, sparsity, etc.).

For the heterogeneity approach, this thesis explores heterogeneous DNN accelerators (HDAs), which contains multiple sub-accelerators that contain different amount of hardware resources and run different dataflows. For the HDA-based approach, this thesis proposes a comprehensive HDA optimization framework, *Herald*, that automatically explore optimization opportunities of mapping DNN layers to a sub-accelerator with the lowest EDP at run time and proper hardware resource partitioning at design time.

Finally, this thesis performs preliminary study on the metric of mapping flexibility and present a preliminary version of formalization in appendix. By this approach, we envision that we can quantify the degree of mapping flexibility of flexible accelerators, which will enable us to explore the trade-off among the degree of flexibility, computational performance, and energy efficiency.

CHAPTER 1

INTRODUCTION

The recent resurgence of deep learning based on deep neural networks (DNNs) facilitated the automation of algorithm design processes for complex regression and classification problems [1, 2, 3, 4, 5, 6, 7]. In addition to the automation, DNNs have shown the capability to drive high quality results even surpassing humans [8, 9, 10] for image recognition [11], speech recognition [12], language translation [13], and many more problems. Therefore, DNN became one of the most valued applications deployed across the cloud and IoT platforms as the backbone of many applications. However, one of the major challenges of DNNs is the high computational overhead, which requires billions of computations for a single run [2, 14, 15] with millions of parameters.

To cope with the challenge, spatial DNN accelerators, specialized hardware for DNN computation, have emerged as a solution. DNN accelerators employ hundreds [16] to thousands [17] of small compute units with small local scratchpad memory termed as processing elements (PEs) to exploit parallelism in DNN computation, which provides computational performance. Also, DNN accelerators employ custom memory hierarchy mainly built with scratchpads to maximize data reuse and custom network-on-chip (NoC) to enable efficient data movement, which provides energy efficiency.

Although DNN accelerators are designed to provide high computational [1] and energy efficiency, the mapping strategies of target DNN layers on an accelerator significantly affects the resulting performance and efficiency [16, 18, 19]. As illustrated in Figure 1.1, a mapping strategy consists of (1) how we size the computation/data tiles (tile sizing), (2)

[1] In this thesis, we distinguish computational performance (i.e., latency and throughput) and operational performance (e.g., the accuracy of the results) since performance is an overloaded term from architecture and machine learning communities; Performance in architecture is throughput or latency while it is accuracy of DNN models in machine learning.

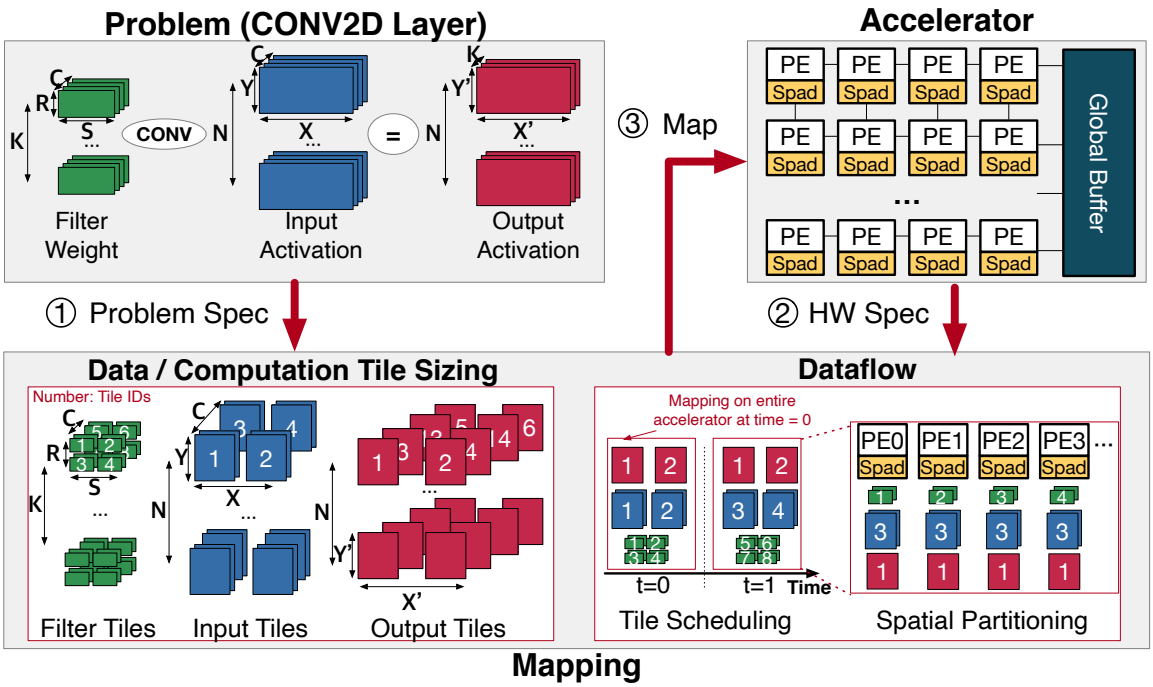


Figure 1.1: Mapping high-dimensional computation/data onto 2D/3D PE array. We present CONV2D operation for example.

how we order the execution of computation/data tiles (temporal scheduling) and (3) how we map computation tiles (and required data accesses) across PEs (spatial scheduling or spatial partitioning). The last two components are collectively referred to as *dataflow* in the accelerator literature [16, 20, 21, 22], which describes the layer-independent essence of a mapping strategy. By adding tile sizes to dataflow, we obtain an instance of dataflow, or *mapping*, which contains all the information to specify a computation schedule in operation level that can be executed by an accelerator.

Mapping dictates the energy consumption of a DNN accelerator since (1) the mapping determines the actual data movement and staging patterns, termed as *data orchestration* in DNN accelerators [16, 18, 19], and (2) the data movement energy dominates, as shown in Figure 1.3. Therefore, optimizing a mapping and providing proper hardware support for the optimized mapping have been prime goals of DNN accelerator design[16, 24, 21, 25]. However, the optimal mapping heavily depends the layer operation and sizes, so no single mapping is ideal for all the layers. This imposes a great challenge to DNN acclerator designs because the machine learning domain is evolving at exponential rate [26] and the layers in

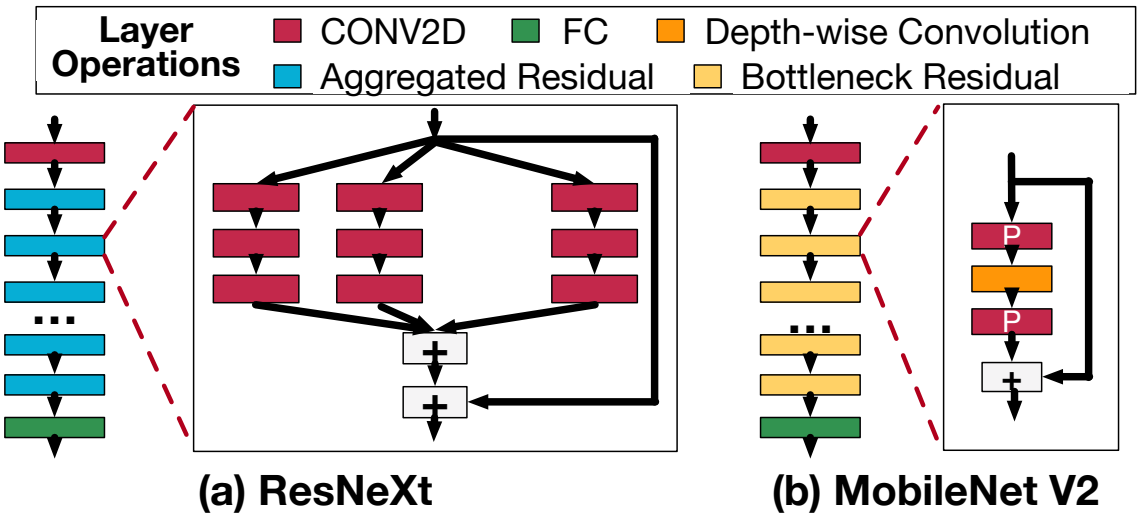


Figure 1.2: Diverse layer operations in two recent DNN models, ResNeXt [6] and MobileNetv2 [5]

Figure 1.3: Energy per bit for each unit activity in a DNN accelerator, reported in [23], based on TSMC 45nm technology. Red texts above each bar show relative energy consumption compared to the computation energy.

DNN models are diverse in operations, as shown in Figure 1.2, and sizes. In addition, DNN accelerators are often designed to support limited dataflow styles as a part of specialization to maximize efficiency for target DNN models at the time of design, which are deprecated soon after the deployment or even during the design time.

Therefore, to address such a challenge, this thesis proposes to design flexible dataflow accelerators that support various dataflow styles and mappings so that DNN accelerators can adapt to new DNN models or layer diversity (or, heterogeneity) in operation and size within a model. As approaches, this thesis first proposes an analytical cost-benefit analysis framework of mappings on DNN accelerators, MAESTRO [19], which quantifies the costs and benefits of mapping choices on flexible dataflow accelerators for each layer in

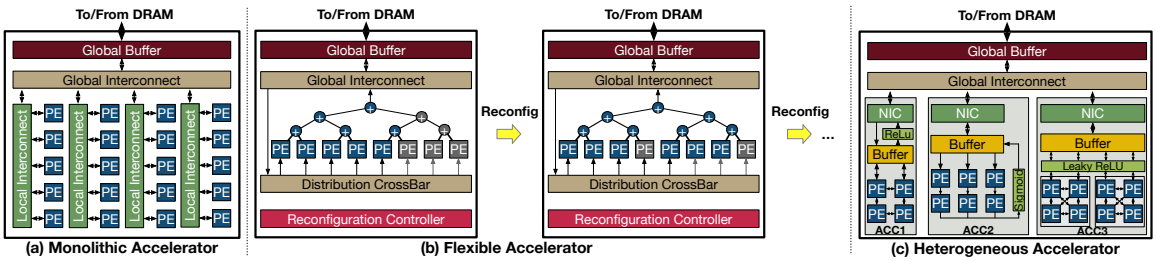


Figure 1.4: Accelerator design options. (b) and (c) are flexible accelerator options this thesis explores. (a) a monolithic accelerator that supports only one dataflow style. (b) a reconfigurable accelerator that can change mapping to run on the same accelerator substrate. (c) a heterogeneous accelerator that consists of multiple sub-accelerator that runs single dataflow style.

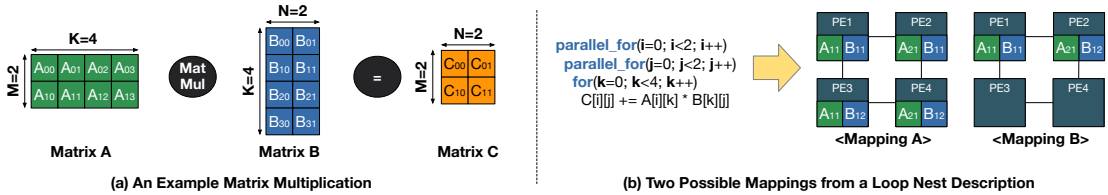


Figure 1.5: The challenge of ambiguous mapping target. (a) An example matrix multiplication operation. (b) A possible mapping description in loop nest, and two possible mappings from the description.

target DNNs. Based on the promising gains we observe using MAESTRO, which will be discussed in Chapter 3, this thesis explores two approaches for designing flexible dataflow accelerators: (1) reconfigurable accelerators, which are runtime-reconfigurable accelerators for various dataflow styles and (2) heterogeneous accelerators, which consist of multiple sub-accelerators that run different dataflow styles. Figure 1.4 (b) and (c) shows an example of each approach, and Figure 1.4 (a) illustrates an accelerator that runs only one dataflow style utilizing entire hardware resources (i.e., monolithic accelerator). In the rest of this chapter, we highlight the challenges for each approach in Section 1.1, summarize solutions this thesis proposes in Section 1.2, discuss related works in Section 2.6, and provide an overview of the thesis in Section 1.5.

1.1 Challenges

1.1.1 Cost-benefit Analysis of Mappings

Although loop nests can precisely describe most of mappings, the representation imposes some challenges for cost-benefit modeling of mappings.

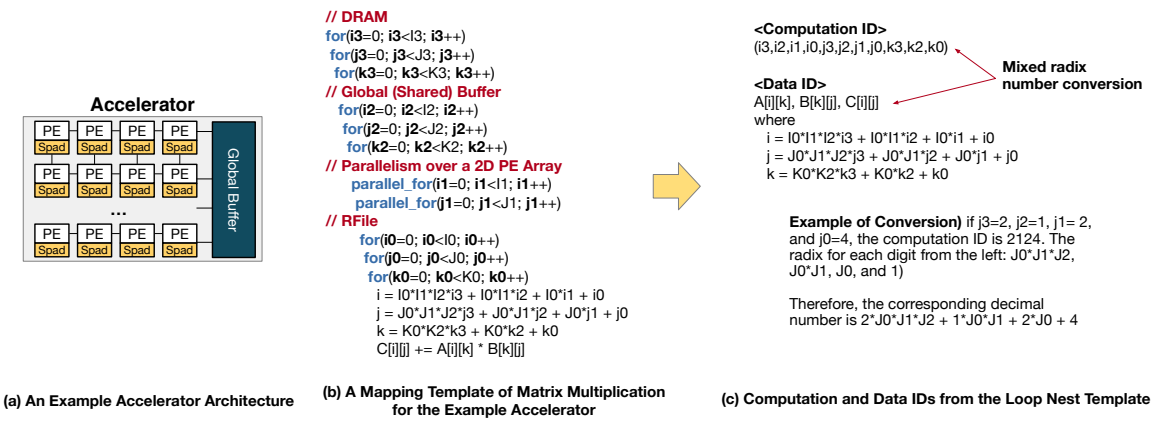


Figure 1.6: The challenge of inferring data accesses and reuse with a matrix multiplication example. (a) An example accelerator. (b) A mapping template of matrix multiplication on an accelerator with 2D PE array with two-level on-chip memory hierarchy based on Timeloop [18]'s loop nest representation style. (c) Conversion of loop indices (computation ID) to corresponding data indices.

Mapping Target of Parallel For Loops. Loop nests utilize parallel for loops to specify the spatial partitioning of computation across PEs [27, 18]. However, parallel for loops but does not specify the exact mapping target, as shown in the example in Figure 1.5. Figure 1.5 (a) shows an example matrix multiplication, and Figure 1.5 (b) shows a possible mapping description in loop nest form. However, the loop nest can indicate either mapping A or mapping B since the mapping target (e.g., Rows/columns of the PE array) is not clearly specified. Based on the context of operation, we can recognize the mapping A is the valid mapping style, but it is challenging for an automated analysis tool to understand it. Defining the semantics to always refer to mapping A style also does not solve this problem since mapping B style can also be a valid mapping (e.g., within sliding window of a CONV2D operation we discuss in Section 2.5, input activation and filter data indices need to change simultaneously like mapping B).

Implicit Data Orchestration. Inferring data accesses and estimating data reuse, which is the key aspect for cost-benefit analysis of DNN accelerators, are challenging for automated analysis frameworks from the computation-centric representation, loop-nest. Figure 1.6 shows an example of such an aspect. Figure 1.6 (b) is a generic mapping template loop nest [18] for a matrix multiplication operation on an example accelerator architecture in Figure 1.6 (a). In loop nests, each combination of loop indices specify one instance

of the loop body (i.e., computation in single-statement loop body). That is, by listing the loop indices, we can specify the ID of one computation in a loop nest. However, data indices need to be inferred from the loop indices via extra computation in the loop body of Figure 1.6 (b), which is equivalent to the conversion of mixed precision numbers to decimal numbers, as shown in the example in Figure 1.6 (c). That is, data orchestration is implicit in loop nest representations, although data orchestration is the key aspect for cost-benefit modeling. Furthermore, inferring data orchestration from compute-centric representation (loop nest) is more challenging for DNN operations because DNN operations often involves high-dimensional data (e.g., 4D tensors in CONV2D operation with 7D computation space).

1.1.2 Reconfigurable Accelerators

Reconfigurable accelerators refer to accelerator substrates that provide partial programmability for running various dataflow and mapping styles, but not Turing-complete programmability like general purpose processors, or GPUs. However, running different mapping styles result in diverse data communication pattern within an accelerator, and supporting such various traffic patterns imposes two challenges: (1) designing light-weight and flexible on-chip network (or, network-on-chip; NoC[2]) to support various DNN accelerator traffics from mappings and (2) maintaining high compute unit utilization for any mapping.

Light-weight and Flexible NoC for DNN Accelerators. A naive approach to design on-chip network for reconfigurable DNN accelerators might result in general all-to-all on-chip network (or, network-on-chip; NoC) such as mesh, tailored for uniform random traffic in general purpose multi-core processors. However, a mesh router [28] is 2.9X larger than a PE [16], and mesh NoC provides poor scalability in both performance and hardware cost [29]. Therefore, designing a light-weight NoC while providing sufficient flexibility and bandwidth for DNN accelerator traffic patterns from various mapping is crucial for implementing a reconfigurable accelerator, which the thesis discusses in Chapter 4.

[2]NoC refers to all possible forms of on-chip interconnection such as bus, crossbar, and so on, not limited to mesh.

Compute Unit Utilization for Various Mappings. Based on mappings, compute unit utilization can be significantly affected. For example, if the M and N dimensions of the matrix multiplication example in Figure 1.5 (a) are one and four, respectively (M=1, N=4), and we follow mapping A style in Figure 1.5 (b), then only the half of the PEs (the first row of the 2x2 PE array) are utilized by the mapping. However, if the PE array was organized as an 1x4 PE array, all the PEs can be utilized by the mapping. Such discord in PE array dimensions and mapping is common in DNN accelerators since hardware is fixed after deployment but the problem dimensions dynamically change, yet the utilization is critical for computational performance and energy [30]. Therefore, designing a reconfigurable DNN accelerator architecture that can provide near 100% PE utilization for any mapping is one of the key research question for reconfigurable accelerator approach, which this thesis discusses in Chapter 5.

1.1.3 Heterogeneous Accelerators

Heterogeneous DNN accelerators (HDAs) employ multiple sub-accelerators running different mapping styles as illustrated in Figure 1.4 (c). HDAs run the best sub-accelerator for each DNN layer to maximize the efficiency. However, although HDAs can provide potential benefits by such micro-specialization for each layer, naive HDA designs and schedulers can lead to inefficiency because of two challenges.

Reduced Parallelism for Each Layer. Given the same number of PEs between a monolithic and an HDA, sub-accelerators in the HDA has smaller number of PEs than the monolithic accelerator since hardware resources need to be distributed (or partitioned) for each sub-accelerator. Therefore, the maximum degree of parallelism each sub-accelerator can exploit for a layer can decrease compared to a monolithic or flexible accelerator. That can lead to higher energy consumption since the amount of data reuse can also decrease (depending on the mapping) if the degree of parallelism decreases.

Determining Layer Execution Schedule. Because multiple sub-accelerators exist in an

HDA, scheduling of layers that determines sub-accelerator-layer matching and ordering of the execution is critical for overall efficiency, which is a problem did not exist in monolithic accelerators. Optimizing the schedule is challenging not only because of the complexity of the scheduling problem, but also various hardware constraints such as memory size or bandwidth need to be considered together. That is, optimal schedule depends on the hardware resources, which leads to a hardware-software co-optimization problem.

1.2 Thesis Contribution

This thesis addresses the challenges we discuss in Section 1.1 using data- and communication-centric approaches, acknowledging that data orchestration is the critical aspect of DNN accelerators to optimize. In addition to the solutions, this thesis also demystifies core concepts to understand costs and benefits of mapping choices on flexible accelerators.

1.2.1 Understanding dataflow, mapping, and data reuse

DNN operations are often high-dimensional problems (e.g., CONV2D operation involves six or seven dimensions), so the mapping of DNN operation and its implication is difficult to understand. In addition, the operand tensors have coupled dimensions in a complex manner, which makes data reuse analysis more difficult. Therefore, this thesis provides backgrounds in DNN operations and accelerators and demystifies the relationship between mapping and data reuse in Chapter 2.

1.2.2 MAESTRO:A Data-centric Approach to Precisely Describe and Model the Costs and Benefits of Mappings on DNN Accelerators

Previous works [16, 21, 18, 27] used loop nests and loop transformations to describe mappings. Although loop nests are able to precisely describe most of mappings (with extra annotations in some cases), complex loop transformations for advanced mappings often result in complex loop nests. Also, modeling data reuses from loop nests that describe

computation, require extra analysis to extract data movement and reuse information, which can be expensive in computation, as we discussed in Section 1.1.1.

Therefore, we present a data-centric description of dataflows and develop a microarchitectural cost-benefit model, MAESTRO, that directly receives data-centric descriptions of a dataflow and reports various statistics quickly (e.g., 0.43 second for analyzing Resnet50 run on a 256-PE NVDLA style accelerator using a laptop with i9-9880H processor with 16 GB memory) [19]. Such a light-weighted cost model was enabled by the aspect that data reuse is explicit in data mapping space, and date-centric description directly specify the data mapping space, not computation space like loop nests. We also implement a hardware design space-exploration tool based on the cost model, MAESTRO, and perform case studies that provides insights about the trade-off space of DNN dataflow and hardware choices for various DNN models. Using the cost model, we show that no single mapping is ideal for all the layers in DNN models, and no single hardware is ideal for all the mappings and layers. We discuss details in Chapter 3.

1.2.3 Microswitch NoC: A Light-weight NoC Specialized for DNN Accelerators

For reconfigurability solutions, our key observation is that the flexible on-chip communication infrastructure enables flexible dataflows. Therefore, we explore flexible a network-on-chip (NoC) design, Microswitch NoC [31], specialized for DNN traffic to minimize hardware overhead while providing sufficient flexibility for any dataflow.

To specialize the NoC for DNN accelerator traffic, we first analyze the traffic patterns from recent DNN models and dataflows. We categorize three traffic patterns and propose a specialized NoC design that employs an array of light-weight switches that distribute communication across the switches, like accelerators try to distribute computation across PEs. We present details about the architecture and evaluation results in Chapter 4.

1.2.4 MAERI: A communication-centric approach for Designing Flexible DNN Acclerator

We explore further flexibility via reconfigurable NoC to support not only various dataflows but also irregular neuron sizes result from various optimizations such as sparsity. For such cases, we observe that irregular sizes of reduction operations can introduce significant compute unit underutilization. We solve the challenge via a new topology of NoC and offloading reduction operation to the NoC. Based on the new NoC that performs in-network processing, augmented reduction tree (ART), we build a full accelerator that employs tree-based NoCs for both of data distribution and reduction, named MAERI [22]. We describe the microarchitecture, routing algorithm, and evaluation results in Chapter 5

1.2.5 Herald: A framework to optimize heterogeneous DNN accelerators

In addition to reconfigurability-based approaches we present in this thesis, we also explore another option for flexibility via heterogeneity. A heterogeneous DNN accelerator consists of multiple sub-DNN accelerators that has different hardware configurations and run different dataflows. We explore optimization opportunities via scheduling layers to the most efficient sub-DNN accelerator. Also, we enable layer level parallelism via multiple sub-DNN accelerator arrays targeting systems that require to run multiple DNN models at the same time (e.g., autonomous cars, AR glasses, or VR headsets).

In Chapter 6, we present the optimization framework, Herald [32], for heterogeneous DNN accelerators. We discuss details about the automatic hardware resource partitioning framework, layer scheduler, and quantified costs and benefits of the heterogeneity-based approach.

1.2.6 Flexibility Metric

Mapping flexibility refers to the capability to run multiple mappings on a single accelerator chip. However, we currently lack methods to quantify the flexibility to compare the degree of flexibility of flexible accelerators. Therefore. we develop a metric for mapping flexibility

and present a preliminary version of formalization of the metric in Appendix Chapter A.

1.3 Thesis Impact

The contributions of this thesis led to many follow-on Works, honors, tutorials, adoptions, and a book.

Honors. MAESTRO(Chapter 3) is selected as Top Picks Computer Architecture in 2019, and MAERI(Chapter 5) received an honorable mention in Top Picks in Computer Architecture in 2019. In addition, original papers of MAERI (ASPLOS 2018 [22]) and MAESTRO (MICRO 2019 [19]) are cited 102 times within two years and 16 times within eight months, respectively, which shows the impact of two works.

Follow-on Works. Yang et al. [33] employs Herald(Chapter 6) and MAESTRO(Chapter 3) to perform neural architecture search on heterogeneous accelerators. Sigma [34], which received the best paper award at HPCA 2019, extends MAERI(Chapter 5) to implement a scale-out accelerator for sparse workloads. Marvel [35] adopts the data-centric approach and cost model of MAESTRO (Chapter 3) and proposes a mapping optimization framework for DNN accelerators. mRNA [30] proposes a mapping optimization framework for MAERI STONNE [36] is a architecture simulator that models MAERI(Chapter 5). Guirado et al. [37] uses MAESTRO to characterize the bandwidth requirement in DNN accelerators In addition to those published works, five more follow-on works are in submission at the time of thesis writing, including a BLAS extension of MAESTRO in collaboration with Sandia National Lab and Pacific Northwest National Lab.

Tutorials. MAESTRO and MAERI led to two tutorials at ISCA 2018 [38] and HPCA 2019 [39], which were attended by more than 30 researchers over the world. Another tutorial based on MAESTRO is scheduled at MICRO 2020 [40] at the time of writing this thesis.

Adoptions. We made MAESTRO(Chapter 3), MAERI(Chapter 5), and Microswitch NoC(Chapter 4) publicly available as open-source projects. The web pages of the open-

source projects are actively accessed by researchers all over the world. For example, MAERI and MAESTRO web pages are accessed 3559 and 4356 times, respectively. In addition to the web pages, source code repositories are also actively accessed. Within two weeks until the time of thesis writing, the open-source repository of MAESTRO[3] and MAERI[4] was viewed by 519 and 105 people, respectively. Both works are widely adopted in academia (Georgia Tech and Cornell), national labs (Sandia National Lab and Pacific Northwest National Lab), and industry (Facebook, NVIDIA, Siemens, and so on).

This thesis also led to many funded research projects. MAESTRO directly led to a $0.5M funding from NSF[5], and MAESTRO and MAERI collaborately led to a 5.5M$ project[6] with Sandia National Lab and Pacific Northwest National Lab.

Book. The data- and communication-centric approach of this thesis led to a text book publication, *Data Orchestration in Deep Learning Accelerators* [41], which will be added to one of the recognized textbook series, Synthesis Lectures on Computer Architecture, published by Morgan & Clay.

1.4 Thesis Statement

The performance and energy efficiency of DNN accelerators depend on mapping of DNN layers on accelerator architectures; by providing mapping flexibility in DNN accelerators via reconfigurability or heterogeneity, DNN accelerators can adapt to any layers and provide optimized performance and energy efficiency.

1.5 Thesis Overview

This thesis is organized as follows:

[3]https://github.com/georgia-tech-synergy-lab/maestro
[4]https://github.com/georgia-tech-synergy-lab/MAERI
[5]https://www.nsf.gov/awardsearch/showAward?AWD_ID=1909900
[6]https://www.news.gatech.edu/2019/11/05/national-labs-georgia-tech-collaborate-ai-research, 2019

- **[Background and Related Works]** In Chapter 2, we discuss backgrounds and related works for this thesis.
- **[Motivation for Flexibility]** In Chapter 3, we discuss data-centric dataflow description and cost modeling (contribution discussed in Section 1.2.2). We highlight the motivation towards flexible dataflow in DNN accelerators Chapter 3 and explore two directions for flexibility: reconfigurability and heterogeneity.
- **[Flexibility via Reconfigurability]** In Chapter 4, we discuss light-weight NoC designs that enables flexible communication in DNN accelerators. In Chapter 5, we discuss a reconfigurable DNN accelerator design that provides high efficiency for irregular DNN computations via reconfigurable in-network processing reduction network.
- **[Flexibility via Heterogeneity]** In Chapter 6, we discuss the costs and benefits of heterogeneous DNN accelerators (HDA), which includes multiple sub-accelerators that contain different amount of hardware resources and run different dataflows. We present an optimization framework for HDAs that performs automatic hardware resource partitioning across sub-accelerators and optimal scheduling of layers on the sub-accelerator array.
- **[Metric for Flexibility]** In Appendix Chapter A, we present a metric to quantify the degree of flexibility in flexible DNN accelerators and discuss how hardware choices constraint the flexibility.
- **[Conclusion and Future Works]** In Chapter 7, we conclude this thesis with summary of our contribution and future works.

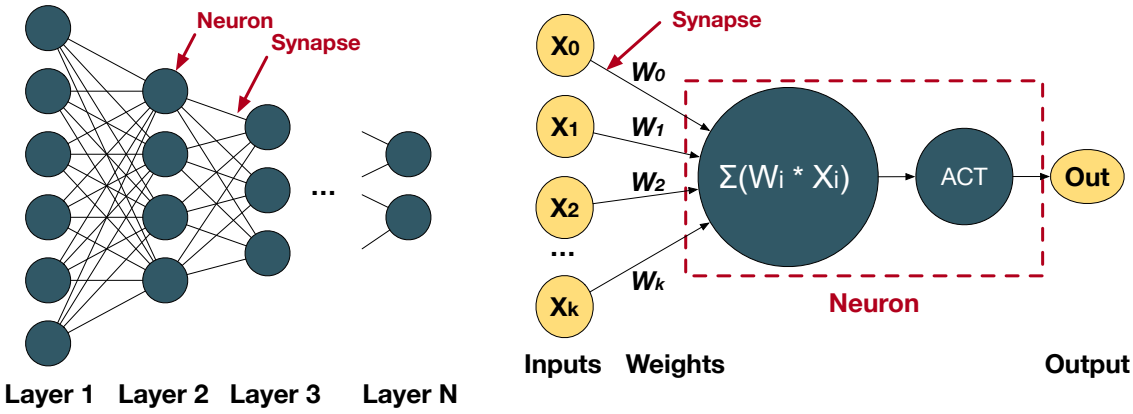
CHAPTER 2
BACKGROUNDS AND RELATED WORKS

In this chapter, we demystify core concepts to understand the impact of mapping on data reuse, and eventually the computational performance and energy efficiency. We first discuss backgrounds in deep neural networks in Section 2.1 and DNN accelearators in Section 2.2. Based on the backgrounds, we clarify the data reuse opportunities and how they are exploited in DNN accelerators using a simple example operation, CONV1D, in Section 2.3. Finally, we provide a deep-dive case study of data reuse in one of the most common practical DNN operations, CONV2D, in Section 2.5.

2.1 Deep Neural Networks (DNNs)

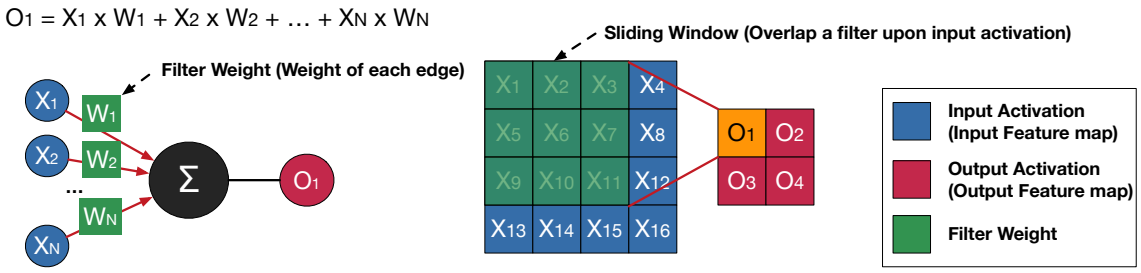
Neural networks are a rich class of algorithms that can be trained to approximate the behavior of complex mathematical functions. They are inspired by human brains and comprise of a large collection of artificial *neurons* connected with *synapses*, as shown in Figure 2.1 (a) and Figure 2.1. The neuron, illustrated in Figure 2.1 (b) is the fundamental building block of DNNs. It computes a weighted-sum to incorporate various features of inputs in different degrees to generate an output value, and applies a non-linear activation function to the output value to generate the final output. The degree of consideration for each input feature depends on the weight on the corresponding synapse.

Each neuron is connected with many other neurons and its output enhances or inhibits the actions of the connected neurons. These connections are termed as synapses, and each synapse has a *weight* associated with it, as illustrated in Figure 2.1 (b). Using neurons as building blocks, neural networks are architected as a series of layers. The first (*aka* input) layer receives the external inputs (*aka* activations), followed by multiple internal (*aka* hidden) layers, followed by the final (*aka* output) layer that provides the final output for the



(a) Deep Neural Network Overview **(b) Neuron Structure**

Figure 2.1: Deep neural network and the structure of a neuron. Each connection in (a) represents a weight. The connectivity can change depending on the type of DNN. ACT refers to an activation function that models non-linearity of the target function.



(a) Neuron Structure **(b) Neuron in a 2D Convolution Layer**

Figure 2.2: (a) The fundamental neuron structure in DNNs, which performs weighted sum. Each edge has weight value multiplied with each input activation, or input feature map. Those multiplication results termed partial sums are accumulated to generate an output activation, or output feature map. (b) An example of neuron in a convolution layer. The region shaded in green over input activation is the filter weight applied over the overlapped input activation, which generates the first output activation highlighted in purple.

function being approximated. As recent neural networks increased the number of hidden layers, the resulting neural networks are called "deep" neural networks or DNNs.

DNNs invovle two classes of operations: training (backward pass) and inference (forward pass). Training is a process to identify weight values on each synapse that provide the best approximation of the target function. Gradient descent methods are the most popular method for training, which propagates errors from the output to input layer (i.e., backward) and update weights to minimize errors. Inference is a process to predict the output for a new input, computing the weight sum of all neurons from the input layer through hidden layers

Table 2.1: Memory Size* and Computations in Selected Recent DNN Models. [1]

DNN Model	Application	Inputs (MB)	Weights (MB)	Outputs (MB)	Computations (GMACs)
AlexNet [**alexnet**]	Image Classification	33.98	473.01	167.43	24.08
VGG 16 [**vggnet**]	Image Classification	116.4	1055.6	230.6	14.4
GoogleNet [**googlenet**]	Image Classification	35.46	52.29	28.69	0.68
Resnet50 [**resnet**]	Image Classification	65.8	923.4	138.6	3.3
MobileNet v2 [5]	Image Classification	52.6	33.4	73.7	0.4
YoloV3 [**yolotiny**]	Image Classification	393.0	431.7	775.0	0.2
UNet [**unet**]	Image Segmentation	16269.9	215.4	28591.4	2428.9
DeepSpeech2 [**deepspeech˙2**]	Speech Recognition	120.5	2.71	15.21	1.60
GNMT [**gnmt**]	Language Translation	513.3	1529.9	2038.7	176.6
Transformer [43]	Language Translation	120	642	132	10.03
GPT2 [44]	Language Translation	70	154.12	190	19.26
NCF [**ncf**]	Recommendation System	31.0	8.6	39.7	0.6
DLRM[a] [45]	Recommendation System	0.04	17.8	0.03	0.002

[a]The memory sizes are shown for the MLP component of DLRM. DLRM also includes embedding tables that can be about 100GB in size

* Assuming 8b inputs, 8b weights and 16b outputs.

and the output layer (i.e., forward).

Because the training algorithm evolves rapidly, training is mainly performed by GPUs or clusters in data centers. In contrast, inferences are often performed at edge devices (e.g., smart phone[42]) in stringent performance and energy requirements. Therefore, most DNN accelerators target inference, so this thesis also focuses on the inference operation.

2.1.1 Popular DNN Models

Over the last decade, researchers have proposed many DNN models every year.

We list some of the state-of-the-art DNNs at the time of this thesis that have had a high-impact in Table 2.1 and describe them here. AlexNet [1] consists of five convolutional layers with grouped convolution and three fully-connected layers. Grouped convolution is partitioning channels into several groups to train the model in parallel in a multi-GPU environment. VGGNet [2] proposed deeper convolutional neural network (11-19 weight layers), demonstrating that deeper models can improve the accuracy. GoogleNet [3] presented the Inception operator that consists of branches of convolutions. In each branch, GoogleNet employs 1x1 convolution or pooling to reduce the computation overhead in the following convolutional layer with larger kernel size (3x3, 5x5, etc.). Resnet [14] introduced the skip

connection, which adds a layer's activation value to the output activation of a later layer (identity operation). The skip connection was effective in addressing the vanishing gradient problem during training and increase training speed. Skip connections were later adopted in many other DNN models [5, 6]. MobileNetV2 [5] is a DNN model designed for mobile devices (or, edge devices). It provides good accuracy close to large models (e.g., VGG16) while requiring significantly small memory size and computation. More recently, DNN models based on neural architecture search [46, 47] have been proposed, which optimize DNN models for both accuracy and efficiency (the number of computations).

Most aforementioned DNN models target computer vision applications, but DNNs are also widely used for natural language processing applications. For speech processing workloads, DeepSpeech2 [12] use a collection of recurrent and convolution layers. Recurrent layers are also used in language translation models such as GNMT [13]. However, the attention mechanism introduced by Transformer [43] emerged for language translation networks like GPT2 [44], which provides significant improvements in the quality of translation results. For recommendation workloads, which accounts for a majority of work among DNN applications in datacenters [48], state-of-the-art networks like DLRM [45] heavily rely on embedding layers, which consists of a series of GEMM operations.

2.2 DNN Accelerators

Table 2.1 summarizes recent DNN models widely used in various domains. From Table 2.1, We can observe that DNNs perform billions of computations and require tens to hundreds of MegaBytes of memory (for storing input, weight and outputs). In additions many applications implemented upon these DNNs often demand real-time inference for the quality of service, which requires high computing power. However, inferences are often run on edge devices such as smart phones that have stringent energy constraints, which requires energy-efficiency. Existing options such as CPUs or GPUs are not ideal options (for computing power and energy efficiency, respectively) to achieve both goals, so specialized hardware

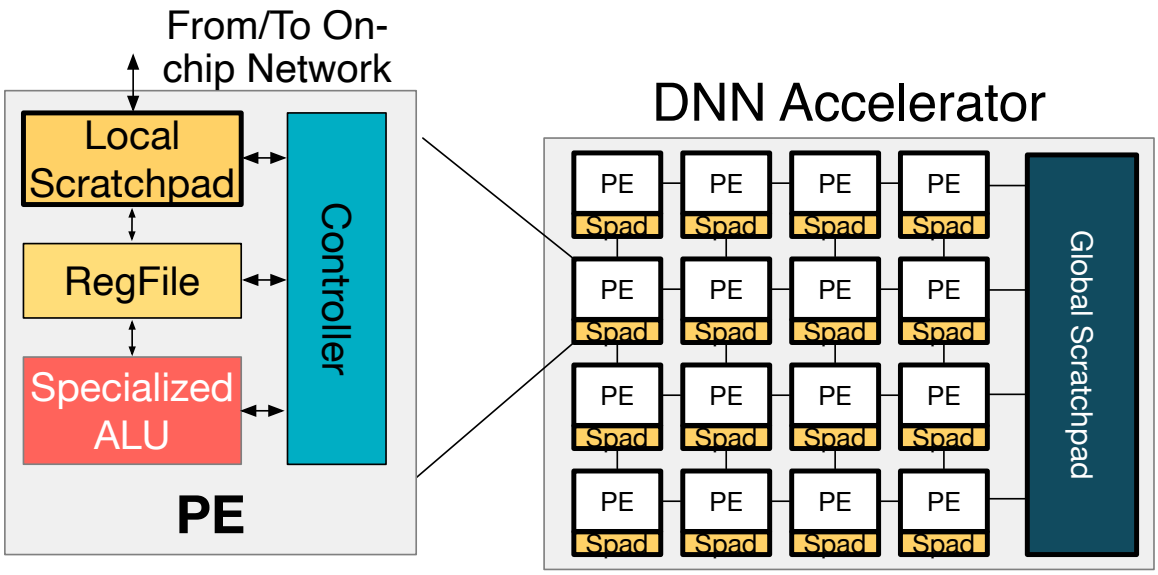


Figure 2.3: An example DNN accelerator architecture.

for DNN inferences, DNN accelerators, are developed.

Figure 2.3 shows an example of DNN accelerator architecture. Many DNN accelerators consist of three components: processing element (PE) array, global (shared) scratchpad buffer, and on-chip network. The processing element is the compute unit of DNN accelerators, which contains a specialized ALU (i.e., only supports operations in target application, DNN), which are often multiply-and-accumulate (MAC) units, register file, (or FIFO) attached to the ALU, and local buffer often implemented using scratchpad memory. The global buffer serves as an intermediate level of memory hierarchy between DRAM and local buffers in each PE, enhancing on-chip data reuse and reducing DRAM acceses. The on-chip network, or network-on-chip, refers to all possible forms of the connectivity among global buffer and PEs, which can be mesh topology shown in the example in Figure 2.3, tree [22], hierarchical bus [16], and so on.

2.3 Data Reuse Opportunities

DNNs heavily rely on weighted sum operations that consist of multiplication and accumulation (MAC) operations. These are abstracted as neurons as illustrated in Figure 2.2. Based on the layer type, the MAC operation can be between vectors, matrices, tensors, or some

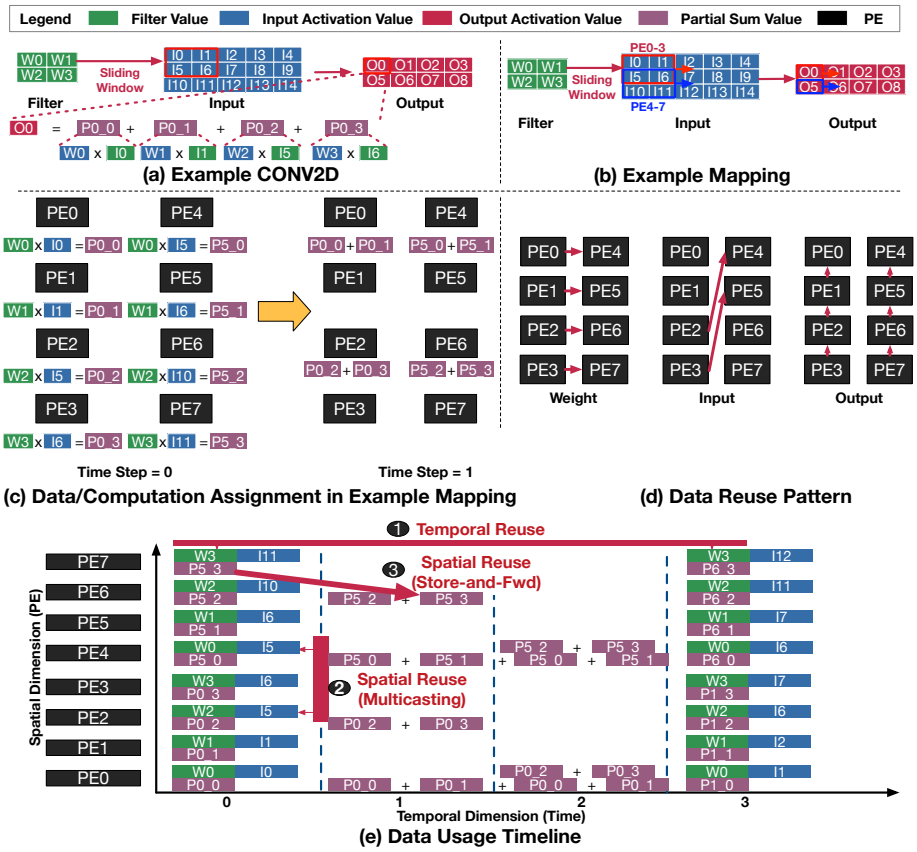


Figure 2.4: Three different data reuse styles in a CONV2D operation with no channels. (a) shows the data labeling convention and the computation required for an output activation pixel. (b) illustrates an example row-stationary data movement pattern [16]. (c) shows temporal and spatial data reuse examples in an Eyeriss-style [16] accelerator with four PEs.

combination thereof. Among many layer types, we focus on convolutional layers as they are heavily used in many DNNs, especially those targeted at computer vision tasks. Convolutional layers perform CONV2D operations that involve tensor-tensor MAC operations, and provide ample data reuse opportunities along multiple dimensions of the tensors. These data reuse opportunities may be *temporal* (across time on the same compute unit, i.e., PE) or *spatial* (across compute units at the same time or across time). The amount of temporal and spatial data-reuse depends on the dataflow and mapping strategy, which we will discuss in Section 2.4.

Figure 2.4 (a) shows an example CONV2D operation and Figure 2.4 (b) illustrates an example row-stationary [16] style mapping for this operation, and Figure 2.4 (c) shows corresponding data and computation assignment on each PEs in the example mapping. We

will discuss CONV2D in more detail in Section 2.5. Figure 2.4 (e) shows all the data points accessed by each PE (y-axis) across time steps (x-axis). We can observe *replications of data points*. Such replication of accessed data points indicates data reuse opportunities revealed by a mapping. We highlight and label some of the temporal and spatial reuse opportunities in Figure 2.4 (e). These refer to the replicated accessed data points across time and space (PE), respectively, in the data usage timeline in Figure 2.4 (e). Spatial reuse opportunity can be further classified as purely spatial (multicasting in Figure 2.4 (d)) or spatio-temporal (store-and-forward in Figure 2.4 (d)). With proper hardware support, we can leverage such data reuse opportunities, and the resulting data reuse pattern of the example is as given in Figure 2.4 (d).

Data reuse opportunities within a DNN accelerator, such as the ones highlighted in Figure 2.4, are actually a subset of data reuse opportunities available within the neural network computation, which we term as algorithmic reuse. Algorithmic reuse from the target workload can be translated into specific data reuse opportunities via three architectural mechanisms within DNN accelerators - staging (i.e., store data in intermediate buffers and load it from the intermediate buffer, not global buffer, in the future), multicasting (i.e., simultaneously sending data to multiple PEs via shared wires; this reduces the number of buffer reads), and forwarding (i.e., sending data to an adjacent PE to use the data in the recipient PE, in the future). The purpose of staging is to keep data points in a local buffer to reuse them in the future, multicasting is to simultaneously send the same data points to multiple PEs to reuse them across PEs, and forwarding to send data points to adjacent PEs so that they can be reused in future iterations. Staging provides *temporal* reuse, while multicasting and forwarding provide *spatial* reuse. These can be implemented via buffer and interconnects respectively. We categorize both multicasting and forwarding as spatial reuse because their data reuse is across space (PEs), and which of the two occurs depends on the implementation choices of the interconnects. We discuss these implementation choices later in Section 3.2.3.

To understand how the DNN computation intrinsically contains algorithmic reuse and how DNN accelerators can translate some of this algorithmic reuse into real data reuse, we provide a simple example, 1D convolution, and analyze algorithmic and actual data reuse of the 1D convolution in the following section.

2.3.1 Data Reuse in 1D Convolution

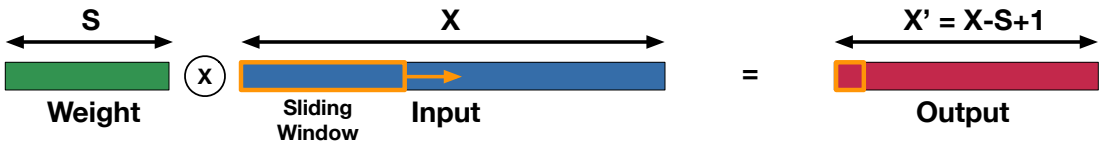
For simplicity, we first analyze the simplest convolution operation shown in Figure 2.5 (a), which is called a 1D convolution or CONV1D operation. CONV1D does not include any height in input activations, height in filter weights, input/output channels (depth), input batches, or activation functions. 1D convolution provides three types of algorithmic reuse, which describe one-to-many relationship from a data point to computations:

- **Input Reuse:** Between adjacent sliding windows, there exist input data points (**halo**) in the overlap of sliding windows. Such input data points are required by computations in two (or more) of the adjacent sliding windows.

- **Filter Reuse:** The same set of filter weight values are required for computing all the output data points.

- **Output Reuse:** One output data point is the accumulation results of element-wise multiplication within a sliding window. We term each multiplication result as a unit partial sum. Since unit partial sums need to be accumulated to one final output value, the intermediate accumulation results can be reused during the accumulation process.

We discuss how this algorithmic reuse can be leveraged in a very simple accelerator with a single processing element (PE), first and then extend the discussion to the multi-PE case. **Single-PE case.** In the 1D convolution example in Figure 2.5 (a), we first place weight filter values corresponding to the first set of input activations in the sliding window. We compute element-wise multiplication within the sliding window and accumulate all the multiplication results to produce one output activation pixel. After an output activation pixel is generated,



(a) 1D Convolution Operation in Output-stationary Style

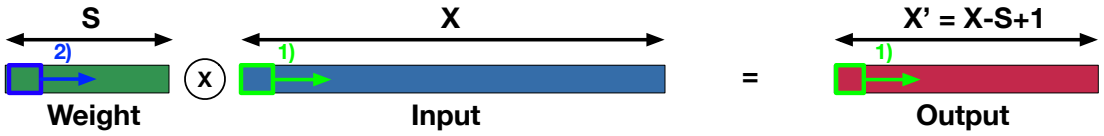
```
for(x' = 0; x' < Bound(X'); x'++)
  for(s = 0; s < Bound(S); s++)
        O[x'] += W[s] * I[x'+s];
```

(b) Output-centric Representation

```
for(x = 0; x < Bound(X); x++)
  for(s = 0; s < Bound(S); s++)
    if(x-s < Bound(X) - Bound(S) && x-s > 0)
      O[x-s] += W[s] * I[x-s];
```

(c) Input-centric Representation

Figure 2.5: (a) A description of 1D convolution operation in sliding window operation over input activation and two loop nest versions of 1D convolution. (b) output-centric and (c) input-centric. Note that both representations are interchangeable. This is an example of an *output-stationary* dataflow.



(a) 1D Convolution Operation in Weight-stationary Style

```
for(s = 0; s < Bound(S); s++)
  for(x' = 0; x' < Bound(X'); x'++)
    O[x'] += W[s] * I[x'+s];
```

(b) Output-centric Representation

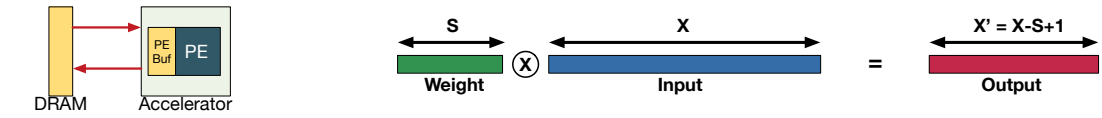
```
for(s = 0; s < Bound(S); s++)
  for(x = 0; x < Bound(X); x++)
    if(x-s < Bound(X') - Bound(S) && x-s > 0)
      O[x-s] += W[s] * I[x-s];
```

(c) Input-centric Representation

Figure 2.6: An alternative style of computation for 1D convolution. Numbers over arrows refer the iteration order. The green boxes moves first; after the green box reaches the end of input feature map, the blue box moves one step. This is an example of a *weight-stationary* dataflow.

we shift the sliding window by one pixel and perform the same computation to generate the next output activation pixel. We repeat this process until the sliding window reaches at the end of the input activation generating all the output activation pixels. Such a computation schedule can be concisely represented using loop nests as shown in Figure 2.5 (b) and (c). Both of the loop nests describe the same computation and schedule but (b) employs output index while (c) employs input index. This is because input and output activation indices are mutually dependent so we need to select one of them to describe the other. We term those two representation styles as input- and output-centric loop nests, respectively.

Note that the loop nest in Figure 2.5 is just one of the possible styles to compute 1D



(a) A Simple Accelerator Model

(b) 1D Convolution Dimensions

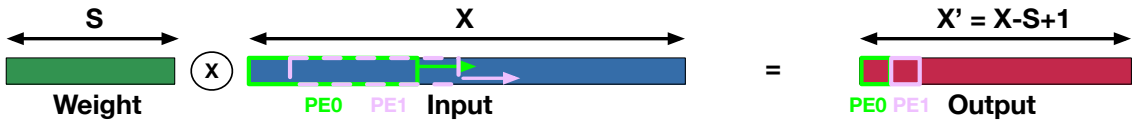
Data Reuse Strategy	Algorithmic minimum DRAM accesses			Maximum operand uses			Minimum PE buffer size for zero re-fetch			Minimum DRAM accesses		
	Weight	Input	Output	Weight	Input	Output	Weight	Input	Output	Weight	Input	Output
Output Stationary	S	X	X'	SX'	SX'	SX'	S	S	1	SX'	SX'	X'
Weight Stationary							1	X'	X'	S	SX'	SX'

(c) Cost Summary

Figure 2.7: The impact of dataflow on memory (PE buffer/DRAM) access and size costs.

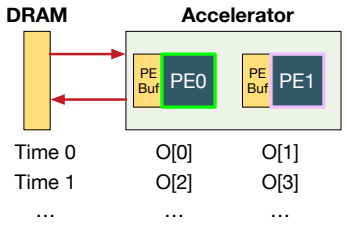
convolution. We call this an *output-stationary* style loop nest, because the output activation index is updated in the slowest manner [16]. Alternately, if we interchange the loop order (loop interchange) in Figure 2.5 (b), we obtain a new version of loop nest as described in Figure 2.6 (b) that generates the same output activation values. In the new loop nest, the weight index changes in the slowest manner because the weight index loop (loop S) is placed at the upper most level. Therefore, this new loop nest is called a *weight-stationary* style. Such combinations of loop transformations are termed as ***dataflows*** [16]. We describe dataflows formally in Section 2.4.

Although both of the loop nest styles compute the same output activation values, they imply different buffer size requirements and buffer access counts which imply dramatically different energy consumption based on the actual dimension size (S and X in the example). Figure 2.7 illustrates this point. For a simple single PE accelerator with a local PE buffer connected to external DRAM running the 1D convolution operation, Figure 2.7(c) summarizes algorithmic reuse, maximum data reuse of each data class, minimum buffer sizes to achieve the maximum data reuse, and minimum DRAM access counts when the maximum data reuse is achieved. The two different dataflow styles (output and weight-stationary) minimizes DRAM accesses of one of the data classes (output and weight respectively) and corresponding PE buffer sizes but have high DRAM accesses and buffer sizes for the other data classes. For example, output stationary matches the algorithmic minimum DRAM access of outputs but does not provide any benefits for weight and input tensors. As the minimum DRAM access numbers show, the overall efficiency of the two dataflow styles



(a) 1D Convolution Operation in Output-stationary Style

```
for(x'2 = 0; x'2 < Bound(X'2); x'2++)
  parallel_for(x'1 = 0; x'1 < 2; x'1++)
    for(x'0 = 0; x'0 < 1; x'0++)
      for(s = 0; s < Bound(S); s++)
      x' = x'2*1*2 + x'1*1 + x'0
      O[x'] += W[s] * I[x'+s]
```



(b) Output-stationary Loopnest on Two PEs (c) Output Mapping on Two PEs

Figure 2.8: An example of dataflow on multiple PEs. (a) shows the 1D convolution sizes with the first data mapping on PE0 and PE1. (b) shows a loop-nest representation of the example output-stationary dataflow. We assume the bound of loop x'0 is 1, for simplicity. (c) describes the parallelism over output activations in the example dataflow.

will depend on the problem dimension (S and X').

Multi-PE case. In previous 1D convolution examples, we analyzed data reuse over one PE over time, or temporal data reuse. When an accelerator has multiple PEs to exploit parallelism as most of spatial accelerators do, the accelerator can have more complex data reuse since the data reuse can occur spatially, or across PEs at the same time, via multicasts. For example, Figure 2.8 shows an output-stationary style dataflow over an accelerator with two PEs in Figure 2.8 (c). Unlike single PE cases in previous examples, this example shows spatial reuse opportunities in input activation implied by the overlapped region (halo) between data mappings on PE0 and PE1 in Figure 2.8 (a). The overlapped data, or halo, imply that they can be multicasted to each PE to reduce DRAM accesses.

2.4 Dataflows and Mappings

In the previous section, we saw how even a simple CONV1D operation can exhibit multiple forms of temporal and spatial reuse depending on the manner in which the computation was tiled, scheduled and partitioned across the hardware resources.

We refer to a specific codification of these choices as a *dataflow*. One such codification

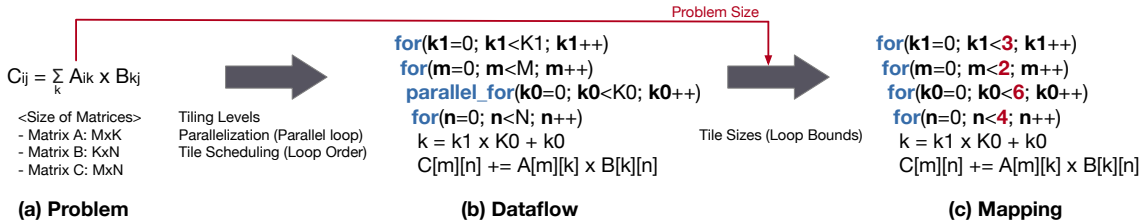


Figure 2.9: Loop nest representation of an example matrix multiplication problem, dataflow, and mapping.

we use is a loop nest representation similar in form to the loop nests we have used in prior sections. The information encoded in this representation is the loop order, choice of loops to parallelize, and the number of loop tiling levels. However, the loop bounds themselves are left as symbolic names. This means that a dataflow prescribes the tiling, scheduling and partitioning strategy, but not the specific *numbers* (or, sizes) of tiles, partitions or loop iterations. Thus, a dataflow expresses an execution strategy for a general problem shape (such as a CONV2D layer) but without explicit workload size information (i.e., size of each data dimensions in input/output tensors of the workload).

A *mapping* is a specific *instance* of a dataflow with numeric (instead of symbolic) loop bounds. Thus, a mapping precisely describes the execution of a specific workload instance, such as the CONV2D layer $VGG16\_3\_2$.

Figure 2.9 shows loop nests that represent an example matrix multiplication problem, an example dataflow, and a mapping based on the problem and dataflow. Figure 1.1 illustrates the overall process of mapping a CONV2D operation onto an example accelerator.

It stands to reason that any reasonable hardware accelerator must have sufficient configurability to support mappings for a variety of workload instances–even though all those mappings may simply be instances of the same dataflow. However, many *flexible* hardware accelerators in fact support multiple dataflows. This is especially true for accelerators with multiple levels in their storage hierarchy. Sections of the dataflow corresponding to the innermost storage levels are often *baked* into the hardware for efficiency, but the outer levels of the hierarchy may consist of programmable state machines, allowing the hardware to run more than one dataflow.

	<Mapping A>	<Mapping B>	<Mapping C>

Mapping	for(x'1=0; x'1< 3; x'1++)
	pfor(x'0=0; x'0< 3;, x'0++)
	for(s=0; s< 6; s++)
	x' = x'1*3 + x'0;
	O[x'] = I[x'-s] * W[s]
		for(x'1=0; x'1< 3; x'1++)
		for(s=0; s< 6; s++)
		pfor(x'0=0; x'0< 3;, x'0++)
		x' = x'1*3 + x'0;
		O[x'] = I[x'-s] * W[s]
			for(x'=0; x'< 8; x'++)
			for(s1=0; s< 2; s1++)
			pfor(s0=0; s0< 3;, s0++)
			s = s1*3 + s0;
			O[x'] = I[x'-s] * W[s]

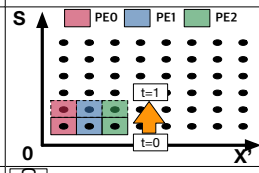
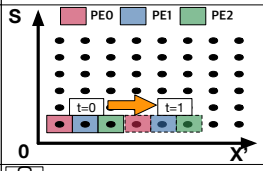
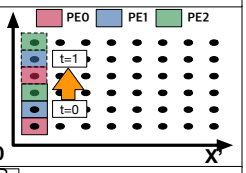
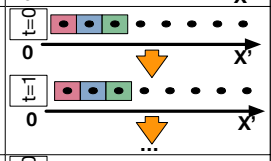
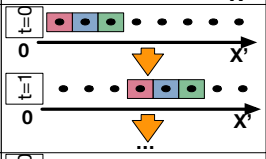
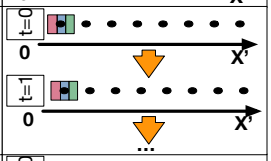
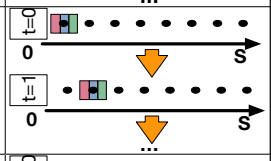
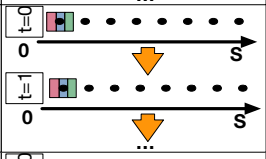
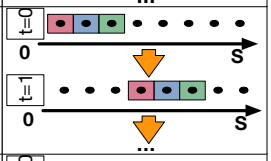
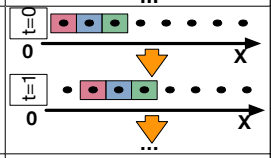
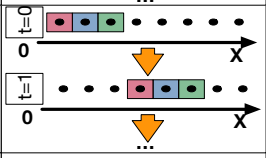
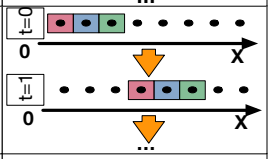
Informal Dataflow Name	Output-Stationary	Weight Stationary	Collaborative Output-Stationary
Iteration Space (Partial sums)			
Output Featuremap Data Space			
Filter Weight Data Space			
Input Featuremap Data Space			
Temporal Reuse	- Temporal reduction of outputs (Output stationary)	- Temporal multicast of filter weights (Weight stationary)	- Temporal reduction of outputs (Output stationary)
Spatial Reuse	Spatial multicast of filter weights	Spatial multicast of filter weights	Spatial reduction of outputs

Figure 2.10: The first three of Six mapping examples for CONV1D operations. The remaining three are shown in Figure 2.11. Mappings A, B, C, and D show the impact of loop order. Mapping E shows the impact of different tile size. Mapping F shows the impact of multi-level parallelism with tiling. **pfor** indicates a parallelized for loop (parallel_for) in this figure.

2.4.1 Deep Dive into Dataflows and Mappings

In Figure 2.10 and Figure 2.11, we build six example mappings upon the simple 1D convolution discussed in Section 2.3.1 to demonstrate how simple changes to a mapping expose various forms of reuse – both spatial and temporal.

The impact of parallelization. Parallel for loops, denoted as pfor in Figure 2.10 and Figure 2.11 specify the spatial data tile distribution across PEs. For example, in mapping A, `pfor(x0=0; x0<3; x0++)` (where `X'` refers to the first dimension of output vector, or output width), spatially distributes indices of the `X'` dimension with a tile size of one across PEs. The number of iterations in a parallel for loop needs to exactly match the total

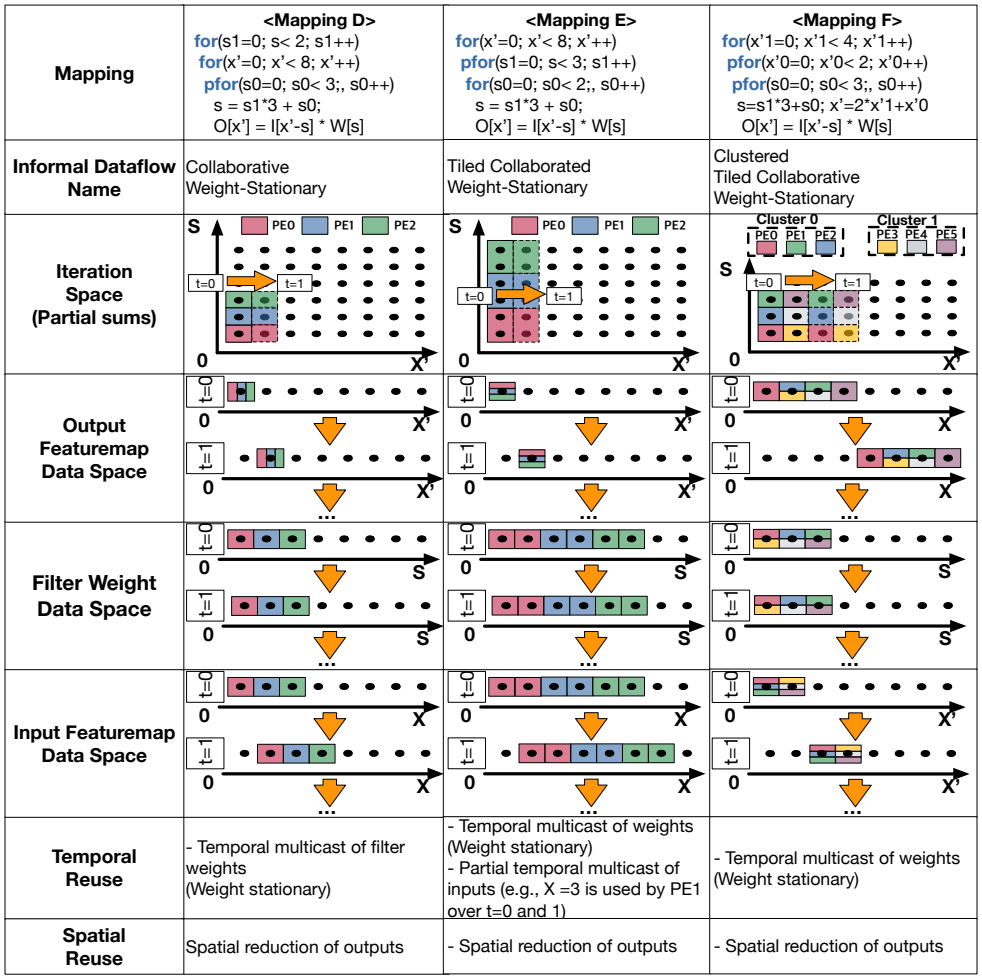


Figure 2.11: The last three of Six mapping examples for CONV1D operations, continued from Figure 2.10

number of PEs. When the loop nest has many parallel for loops, the product of the each loop bound needs to match the total number of PEs.

Similarly, a temporal for loop denoted as plain "for" in loop nests specifies the distribution of a dimension across time steps in each target PE. That is, temporal for maps the same set of indices for a dimension across PEs. For example, `for(s=0; s<6; s++)` of mapping A in Figure 2.10 (where `S` refers to the first dimension, or width, of filter weight data class), distributes indices of the `S` dimension with a chunk size of one across time steps. This temporal distribution can be viewed from the data space of the filter weight data class in the same column of mapping A in Figure 2.10. Since all PEs get same data indices corresponding to a temporally mapped dimension, this may create opportunities for *spatial*

reuse, i.e., spatially reusing the same data values across PEs in a time step.

The impact of loop order. The order of parallel and temporal for loops in a loop nest dictates the order of data movement, which also changes the data mapping to PEs across time. A change in the order can result in an entirely different stationary behavior. For example, the sequence of loops in mapping A in Figure 2.10 (i.e., parallel for on X′ followed by the temporal for on S) indicates that all data indices of S should be explored before working on the next chunk of X′ indices. This order results in temporal reuse data corresponding to X′ indices (i.e., partial sums, for all indices of S) leading to an output stationary dataflow. This behavior can be observed from the iteration space for the mapping A in Figure 2.10.

If the loops over x'0 and s are interchanged as shown in mapping B in Figure 2.10, the dataflow of the resulting mapping now keeps weights stationary because PEs can temporally reuse data corresponding to S indices, (not X′ indices anymore), (i.e., weight values, for all indices of X′) before moving to the next set of S indices. Similarly, mappings C and D in Figure 2.10 and Figure 2.11 show the spatial distribution on S instead X′, and also the impact of data movement order on temporal reuse leads to different stationary dataflows.

The impact of tiling. In all of the mapping styles from A-D in Figure 2.10 and Figure 2.11, the tile sizes are all one – resulting in either no temporal reuse (e.g., partial output sums in case of mapping B) or full temporal reuse (e.g., input feature map in mapping C). Increasing the tile size of the parallel and temporal for loops can help in capturing partial temporal reuse opportunities, or convolutional reuse in input feature map of convolution layers. For example, the parallel for on the S dimension in mapping E in Figure 2.11 enables partial temporal reuse of input feature map data across time steps.

Also, tiling can be used to divide iterations into smaller units as shown in mapping D where the loop over dimension S is split into two loops and placed across loop over dimension X′, which indicates that we do not have to fully explore indices of S dimension before we change mapped indices on X′ dimension. This not only removes fundamental

restriction of loop nests that require all the inner loops complete before updating the loop variable of a loop but also reduces the data tile size to fit into on-chip global and PE buffers.

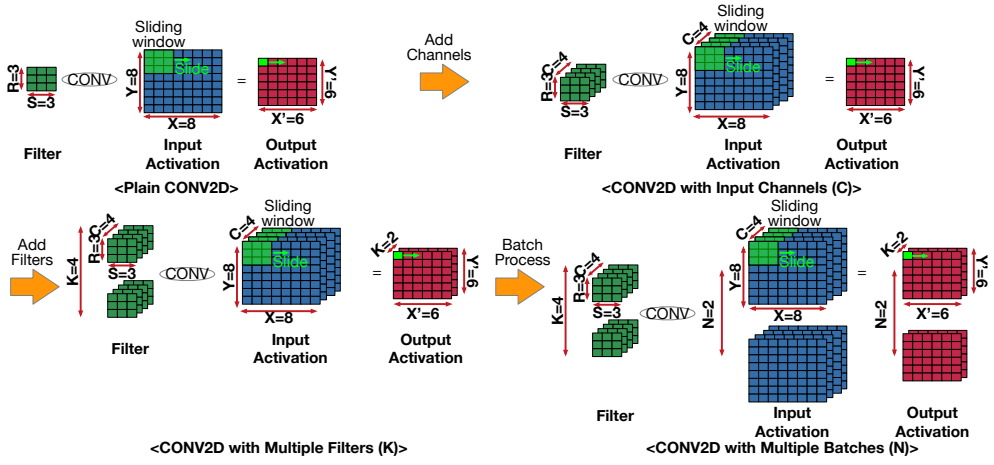
Exploiting multi-dimensional spatial distributions. To exploit multi-dimensional parallelism, a loop nest can employ multiple parallel for loops like mapping F in Figure 2.11. Each parallel for loop indicates a parallelization dimension, which eventually implies dimensionality in a PE array. For example, mapping F in Figure 2.11 has two parallel for loops and implies two clusters of PEs with three PEs in each, as shown in iteration space row of the example. However, if two parallel for loops are specified on an overlapped dimension (e.g., input feature map column and filter column), it does not imply additional dimension in a PE array. A loop nest needs to ensure the product of loop bounds of each parallel for matches the number of PEs. If the numbers do not match, the mapping results in either unavailable to map on a PE array (when the product of loop bounds, or *the degree of parallelization*, is larger than the number of PEs) or PE underutilization (the degree of parallelization is smaller than the number of PEs). This is because the degree of parallelization is the total amount of parallelism implied by the data orchestration and the number of PEs is actually available parallelism in hardware. If the numbers do not match, it implies that the mapping does not properly fit onto the target hardware.

As observed in the examples of dataflows and mappings of a 1D convolution introduced in Figure 2.6, loop order and resulting stationary behavior based on dataflow play a key role to materialize algorithmic reuse into an actual reuse in computing schedule. However, to realize such algorithmic reuse in hardware, appropriate hardware support is required. For example, the network-on-chip (NoC) between the DRAM and the PE array in Figure 2.8 (c) needs to support multicasts to exploit spatial data reuse opportunities implied by the dataflow in hardware. Without the support for multicasts from NoC, the spatial reuse opportunity is forfeited and translated into independent unicasts, which do not reduce DRAM accesses at all in the example architecture of Figure 2.8 (c). To discuss the hardware support for each data reuse type, in the next section, we classify the data reuse in more fine-grained manner

Data Dimension	Batch	Output Channel	Input Channel	Filter Row	Filter Column	Input Row	Input Column	Output Row	Output Column
Notation	N	K	C	R	S	Y	X	Y'	X'

* Upper case: Size, Lower case: index

(a) Data Dimension Labeling Conventions



(b) Constructing the Full CONV2D Operation from Plain CONV2D

```
for(n=0; n<2; n++)
 for(k=0; k<2; k++)
  for(c=0; c<4; c++)
   for(y=0; y<8; y++)
    for(x=0; x<8; x++)
     for(r=0; r<3; r++)
      for(s=0; s<3; s++)
       if(y-r >= 0 && x-s >= 0)
        O[k][y-r][x-s] += W[k][c][r][s] * I[c][y][x];
```

(c) Input-Centric Loop Nest

```
for(n=0; n<2; n++)
 for(k=0; k<2; k++)
  for(c=0; c<4; c++)
   for(y'=0; y'<6; y'++)
    for(x'=0; x'<6; x'++)
     for(r=0; r<3; r++)
      for(s=0; s<3; s++)
       O[k][y'][x'] += W[k][c][r][s] * I[c][y'+r][x'+s];
```

(d) Output-Centric Loop Nest

Data Dim. / Data class	Batch (N)	Output Ch. (K)	Input Ch. (C)	Filter Row (R)	Filter Col. (S)	Input Row (Y)	Input Col. (X)
Output Activation	✓	✓		✓	✓	✓	✓
Input Activation	✓		✓			✓	✓
Filter Weights		✓	✓	✓	✓		

* Output row(Y') = Y-R+1, Output column(X') = X-S+1

(e) Data class and Coupled Dimensions in Input-centric Loop Nest

Figure 2.12: An example of a convolutional layer with its dimensions and indexing are shown in (a), and a visualization of the convolution shown in (b). An input-centric and output-centric view of loop nests corresponding to the convolution is shown in (c) and (d) respectively. A summary of the coupling among dimensions and data classes (tensors) are shown in (e), where a table entry with a check mark indicates that the dimension in the column is coupled with the data class in the row.

and discuss hardware choices to actually exploit the data reuse in the accelerator.

2.5 Dataflows and Data Reuse in CONV2D

CONV2D operation is one of the most common operations in convolutional neural networks (CNN) that involves seven data dimensions across three tensors: input/output activation and weight tensors. Unlike CONV1D, CONV2D operation involves seven data dimensions, which makes the CONV2D operation a high-dimensional operation. Therefore, we first introduce the CONV2D operation and then discuss the data reuse opportunities and dataflow

choices.

2.5.1 CONV2D Operation

Recall that CONV1D computes an output vector (output activation) using two input vectors (input activation and filter) as operands. The CONV2D operation computes an output **tensor** (output activation) using two input **tensors** (input activation and filter) as operands. To understand the high-dimensional CONV2D operation, in Figure 2.12, we construct the full version of CONV2D operation from the simplest form, which we name as plain CONV2D. Figure 2.12(a) lists the labeling conventions for each data dimension.

Plain CONV2D. Assuming single-layer CNN, input and output activation can be viewed as input and output images of an image processing program (e.g., blurring), and filter can be viewed as an image processing mask (or, kernel). Assume that the images are gray-scale images, then the simplest CONV2D operation upon the gray-scale images can be visualized as the plain CONV2D in Figure 2.12 (b). Note that the sliding window is now two-dimensional unlike the CONV1D operation; thus the operation is termed as CONV2D. The 2D sliding window in CONV2D needs to sweep the entire surface of the input activation. That is, the sliding window moves along both of the input row (Y) and column (X) dimensions, performing the same element-wise multiplication and accumulation as we computed in CONV1D operation.

CONV2D with input channels. Just like images typically have red-green-blue (RGB) channels, input activation can also have channels. When input activation has multiple channels, filters need to have the same number of channels as that of input activation, so that all the input activation channels have corresponding filter values. That is, the sliding window is now three-dimensional, as shown in CONV2D with input channels in Figure 2.12 (b). Therefore, the element-wise multiplication results are accumulated not only in filter row and column dimensions but also in the input channel dimensions, visualized as a depth dimension in Figure 2.12 (b). The width and height of the sliding window are referred as

filter row (R) and height (S) dimensions. The width and height of the input activation where the sliding window sweeps are referred as input row (Y) and column (X) dimensions.

CONV2D with input channels and multiple filters. A 3D filter with input channels extracts one feature from the input activation. When we need to extract multiple features, we apply multiple 3D filters. Since we have multiple filters, we obtain multiple and independent set of output activation values, constructing the output channel (K) dimensions in output activations, as shown in CONV2D with multiple filters in Figure 2.12 (b). We can view this as swapping the filters in the previous version (CONV2D with input channels), generating multiple set of output activation values and arranging them as the depth (i.e., channel) of the output activation.

CONV2D with channels and multiple batches. We often process multiple input and output activations during inference. Instead of processing them one-by-one, we can construct a batch (N) of the input activations and run all at once. Such scenario is illustrated in CONV2D with multiple batches in Figure 2.12 (b). This can be viewed as we are running multiple instances of the previous version, CONV2D with input channels and multiple filters, and arranging the inputs and results as the sequence of 3D input and output activation tensors, like illustrated as N dimension in the last step of Figure 2.12 (b). This version is the full CONV2D operation that can be easily found in recent CNNs such as Resnet [**resnet**].

Loop-nest representation of full CONV2D. Using a subset of the data dimensions, we can represent the computation of CONV2D in a loop nest. Figure 2.12 (c) and (d) shows two possible loop nest representation of the example CONV2D operation in Figure 2.12 (b). Two versions exist since the row and column indices of input/output activation tensors are mutually deducible. For example, the input row index (y) corresponds to given output row index (y') and filter row index (r) can be computed $y = y' + r$, as shown in Figure 2.12 (d) (e.g., if we process the first output row (y'=1) and the second filter row (r=2), we need to access inputs in the third row (y = 'y + r = 1 +2 = 3). Based on the choice of input and output row/column indices to use the in loop nest representation, we can obtain two versions

of loop nests. Like we did in the CONV1D examples, we name those two versions input- and output-centric loop nests, respectively.

2.5.2 Data Dimension Coupling and Data Reuse Opportunities

When a data dimension exists in a tensor, then the data dimension is *coupled* with the tensor. Figure 2.12 (e) shows how seven data dimensions in the CONV2D operation are coupled with three tensors.

Data dimensions other than activation row and column do not have such coupling but they can be coupled with multiple tensors or only one tensor. For example, the input channel index `c` appears in both filter and input activation, and the output channel `k` appears in both filter and output activation. We call these dimensions *coupled* to these indices, as the position in the data space changes when the index is modified. That is, when a tensor *A* is independent of a dimension α, the mapping of the tensor is stationary when α is updated in a loop nest. This implies the possibility of temporal data reuse via proper loop order. From another perspective of spatial mapping, when we parallelize the loop on the dimension α, the tensor *A* is stationary across **space** (i.e., PEs). This implies the possibility of spatial data reuse via multicasts.

Likewise, we can exploit the dimension coupling via loop order and parallel loops to expose the data reuse opportunities. In addition, we can also control the amount of data reuse via loop tiling. Since the CONV1D operation discussed in Figure 2.10 and Figure 2.11 does not show complex dimension coupling relationship, we provide a realistic dataflow example in Figure 2.13 and discuss the data reuse in the example.

2.5.3 Data Reuse in a CONV2D Example

Figure 2.13 (a) and (b) describe an example mapping based on the row-stationary dataflow introduced in Eyeriss [16] on an example convolution shown in Figure 2.12 (b). Figure 2.13 (c) describes actual mapping of tensors on an accelerator with six PEs. The six PEs are

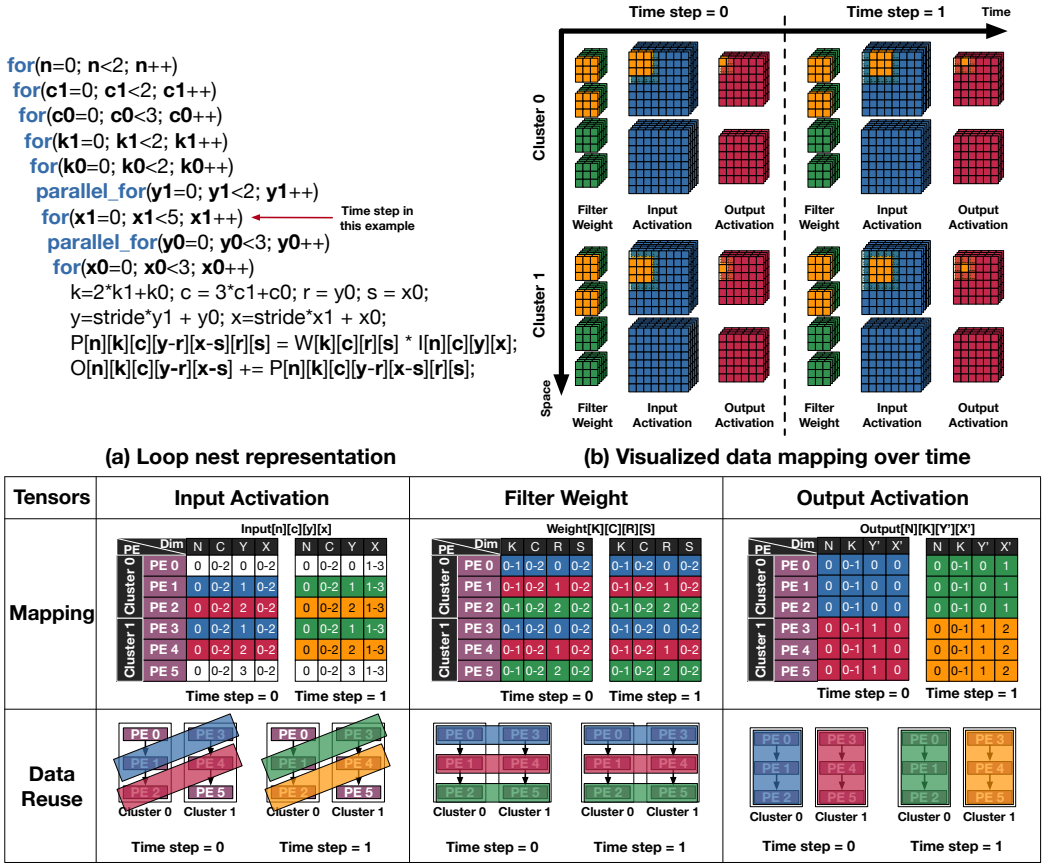


Figure 2.13: Detailed mapping description of an accelerator with six PEs running a mapping based on row-stationary style [16] dataflow. The colors in (b) represent each tensor and a computation tile, and that in (c) represent replicated data (i.e., data reuse opportunities) from the mapping. We refer to computation tile iterations on the PE array in the example accelerator as timesteps in this example to emphasize the temporal scheduling of computation tile mapping. We mark the loop nest corresponding to the computation tile iteration (or, timesteps in this example) in (a).

clustered into two groups of three PEs each because the loop nest has multiple parallel for loops. The clustering dimensionality follows the loop bounds of each parallel for. Note that we refer each computation tile iteration upon the PE array in the accelerator as a time step, which corresponds to the iteration of input activation tile (loop x1), as we denote in Figure 2.12 (a). Therefore, we refer to loop x1 as the unit loop of this example.

The mapping places the loop over activation row tile $y1$ above the unit loop (loop x1), which makes the activation row stationary (i.e., row-stationary) across time steps. We deep dive into the data reuse pattern for each of three tensors: input activation, filter, and output activation tensors.

Input Activation. Each of the PE clusters receives three input activation rows with stride one. The input activation column dimension is fully covered within the inner loops under the unit loop. Also, loops on other coupled dimensions (input channel$c$ and batch n) with input activation are placed in the upper positions of the unit loop. Therefore, those three dimensions (n, c, and x are stationary. However, because we have stride of one in activation row dimension and we parallelize the activation row across PEs within a PE cluster in order, the replicated input activation is skewed as shown in the first column of Figure 2.13 (c). Therefore, we can observe a spatial data reuse opportunity on input activations in a diagonal direction of the PE array.

Filter. Each of the PE clusters has full filter row(r) and column (s), and all the loops on input channel (c) and output channel (k are placed in the outer loops of the unit loop. Because filter is coupled with those four dimensions (k, c, r, s), the filter is stationary across PE clusters. Within a PE cluster, since we parallelize filter rows along $y0$), we distribute three filter rows across three PEs in the PE cluster. This results in horizontal replication of filter values as shown in filter weight column in Figure 2.13, which implies a spatial data reuse opportunity.

Output Activation. Because the batch(n) and output channel (k) dimensions are placed in upper loops of the unit loop, and activation column is fully mapped in the inner loops of the unit loop, those three dimensions (n, k, and x') are stationary across PE clusters. However, the output activation row is not stationary across PE clusters. Each PE cluster receives three consecutive input activation rows, and this can be translated as one output activation row because the filter row size is three. Because of the parallel for on activation (loop $y1$), each cluster receives different output activation row with stride one. That is, each cluster processes different output rows. Within each PE cluster, three PEs collaborate to generate partial sums and spatially accumulate them to produce output activation values. For the accumulation, partial sums flow linearly within each PE cluster, as shown in the third column of Figure 2.13 (c).

2.6 Related Works

Hardware DSE and dataflow optimization: Dataflow optimization is one of the key optimization targets in many recent DNN accelerators such as Eyeriss [16], Flexflow [21], SCNN [20], and NVDLA [49]. C-brain [50] and Flexflow [21] analyzed the costs and benefits of three preset dataflows using analytical models that measure the utilization of PEs and explored the opportunity of adaptive dataflow among those three. Ma et al. [25] also constructed an analytic model for convolutions on FPGAs focusing on three loop transformations; interchange, unroll, and tiling. Although their analytic model provides an intuition for trade-offs of dataflows, the model focus on one weight-stationary dataflow style without considering communication delay in NoCs, which can dominate for dataflows with large tile sizes. Also, the target dataflow is optimized for HLS flow, and it requires to write complex annotated loop nest with HLS synthesis directives. Caffeine [51] proposed a full automatic FPGA flow that includes pragma-based annotation in programs, dataflow optimization framework, and DSE for FPGAs based on the analytic model defined over loop tiling and unrolling. However, the dataflow search space is limited due to fixed loop orders; three presets termed straightforward, input-major, and weight-mapping.

Past works related to data-centric approaches: In chip-multi-processor (CMP) domain, Kodukula et al. [52, 53, 54] explored data-centric approaches to optimize dataflow through traditional memory hierarchy using locality-enhancement transformations such as multi-level data blocking [52] and data shackling [53]. However, those data-centric approaches explored optimization opportunities only in multi-level caches without precisely estimating energy and throughput of input kernels. Therefore, those work cannot be directly applied to accelerators with scratchpad memories and point-to-point communication capability between adjacent compute units.

DNN accelerators for flexible dataflows: FlexFlow [21] and DNA [55] are two recent DNN ASIC proposals with similar motivation as this work. FlexFlow demonstrates a design

that provides feature-map, neuron, and synapse-level parallelism, by different mapping strategies across PE rows and columns. DNA leverages Weight, Input, and Output Reuse within the same fabric. However, both FlexFlow and DNA's flexible dataflows are restricted *within* a layer, not across layers.

NoCs for DNN accelerators: Most of NoC studies for DNNs [56, 57, 58, 59] have proposed meshes due to their flexibility to support all-to-all communication. Diannao [60] and Shidiannao [61] relied on mesh-based interconnects for data transfer. However, meshes add extremely high area and power overheads as our work [22] and others [29] have shown. Dadiannao [62] employed fat tree for scatter and gather and 3D mesh via hyperTransport 2.0 to distribute data among nodes. Eyeriss [63] uses separate buses for its scatters and gathers. Eyerissv2 [27] adopted a hierarchical mesh NoC for flexibility.

CNN accelerators: Convolution Engine [64] explored the trade-off between flexibility and efficiency in accelerator domain with an example of convolution accelerator for image processing domain. Diannao [60], DaDiannao [62], ShiDiannao [61] are early spatial DNN accelerators. A recent work in FPGA [65] proposed constraint aware optimization tool for FPGA based accelerators. The design used simple adder trees within their computation engine, which can benefit from ART [22]. Eyeriss [16] analyzed data flow patterns in existing CNN accelerators and proposed a new data flow pattern for CNN acceleration called row stationary, which performs better than other data flow patterns in terms of throughput and energy efficiency.

Cross-layer CNN Accelerators: A lot of recent works have performed design-space exploration of DNN accelerators, such as finding a better way to map data on to hardware or the best configuration of DNN accelerators, using novel simulation infrastructures [66, 67, 68]. Fused-layer CNN [69] and others [70, 71, 72] have explored CNN architectures optimized for cross-layer optimizations over FPGAs. FPGAs provide immense flexibility in tuning the RTL for the right dataflow pattern and filter size(s) for mapping.

Sparse CNN Accelerators: SCNN [20] is a recent accelerator for sparse CNNs, that

leverages sparsity in activations and weights. Cnvlutin [73] compresses activation values based on the ReLU operator. Cambricon-X uses weight sparsity to keep only non-zero weights in its buffers. EIE [74] uses a compressed representation of weights and activations, delivering only non-zero operands to multipliers, for FC layers.

RNN accelerators: Although the algorithm of RNNs has been discussed for more than 20 years [75, 76, 77, 78], hardware design of RNNs was not as active as that of CNNs. In the last two years, hardware acceleration of LSTMs on FPGAs [79, 80, 81, 82] has been explored. Maeda et al. suggested new learning algorithm of Hopfield RNN [83] which has been implemented on FPGA [84, 85]. An ASIC implementation for accelerating the control of RNN networks was recently demonstrated [86].

DNPU [87] implements CNN and RNN-LSTM in a single chip but requires independent compute units for CNN and RNN. Google's TPU [17] also supports both CNNs and LSTMs. However, the systolic array supports only one dataflow style (a weight-stationary style parallelizing output channel across columns and unrolling filter row/column and input channel across rows) so it cannot adapt to various DNN models.

2.7 Summary

In this chapter, we demystified DNNs, DNN accelerators, and how data reuse is exploited in DNN accelerators running example DNN operations, CONV1D and CONV2D. In our case studies in Figure 2.10 and Figure 2.11, we could observe that data reuse is explicit in data space. Based on this observation, we explore how we can exploit the data space in mapping representation and cost-benefit modeling in the following chapter.

CHAPTER 3

MAESTRO: A DATA-CENTRIC APPROACH TO DESCRIBE MAPPINGS AND MODEL COSTS AND BENEFITS OF MAPPINGS ON DNN ACCELERATORS

The efficiency (performance and energy efficiency) of a DNN accelerator depends on three factors: (1) the workload (DNN layers), (2) the amount and type of available hardware resources (hardware), and (3) the mapping strategy of a DNN layer on the target hardware (mapping). That is, we can predict the efficiency (latency, energy, buffer requirement, etc.) of an accelerator when we have full parameters for those three factors, which can guide the DNN accelerator design for better efficiency.

However, modeling the complex high-dimensional DNN accelerator design space over the three factors is challenging because it requires a deep understanding of complex interaction of hardware components, mapping, and DNN layers. In addition, one critical requirement on the efficiency estimation is that it needs to be fast since the design space (e.g., 480 million valid designs in our hardware DSE even if we fix the target mapping and layer) is huge, and we need to query the efficiency of candidate designs in the search space when we search for an optimal design. How do we implement such a fast efficiency estimation framework that thoroughly considers all the parameters of the three factors that determine the efficiency of DNN accelerators?

To enable such an efficiency estimation framework, we propose to take a data-centric approach since data orchestration (data movement and staging) is the key behavior to model data reuse. We discuss how we can describe mappings from based on a data-centric approach, and how the data-centric mapping description enables fast efficiency estimation framework, MAESTRO.

3.1 Data Reuse in DNN Accelerators

3.1.1 Data in DNNs

We present an example of a multi-channel CONV2D described in Figure 2.1 that involves seven data dimensions across three data structures: input/output activation and weight tensors. Although our approach can be applied to various DNN layers—CONV2D, fully-connected (FC), LSTM, separable convolution, and so on—we focus on CONV2D and its variants in this work because convolutional neural networks (CNNs) are popular, and CONV2D accounts for more than 90% of overall computation in CNNs [88, 16].

CNNs involve three tensors; input activation, filter weights, and output activation[1]. As shown in Figure 2.12 (b), all of three tensors are four-dimensional in full CONV2D, and some of the data dimensions in each tensor overlap, resulting in seven distinct dimensions. Note that input row/column indices can be deduced using output row/column indices and filter row/column indices and vice versa for output row/column indices using the relationship described in Figure 2.12 (y = y'+ r and x = x'+ s).

Also, the input channel index `c` appears in both filter and input activation, and the output channel `k` appears in both filter and output activation. We term such dimensions as *coupled* dimensions, as the position in the data space of the tensors with the coupled dimension changes when the index of the dimension is modified. Because of these specific data access patterns, we can transform loop nests to keep one of the tensors *stationary* over a range of space or time (i.e., unchanged in a local buffer). Such an optimization can significantly reduce global buffer access counts in DNN accelerators, which reduces costly energy consumption from the global accesses.

[1]For conciseness, we use input, filter, and output to refer each tensor.

3.1.2 Data Reuse Taxonomy

We broaden the taxonomy of dataflows by observing that data reuse originates from two behaviors of DNN accelerators over time and space - multicasting (input tensors) and reduction (output tensors).

Multicasting. Spatial multicasting reads a data point from a buffer only once, spatially replicates the data point via wires, and delivers the data point to multiple spatial destinations (i.e., PEs), which reduces expensive remote buffer accesses and saves energy. Likewise, temporal multicasting also reads a data point from a large remote buffer only once, temporally replicates the data point via a smaller local buffer, and delivers the data point to multiple temporal destinations (i.e., different time instances) at the same PE, which also reduces expensive remote buffer accesses and saves energy.

Reduction. Spatial reduction accumulates partial outputs from multiple spatial sources and spatially accumulates them via multiple compute units (e.g., an adder tree). Similarly, temporal reduction accumulates partial outputs from multiple temporal sources (i.e., partial sums computed at different time) and temporally accumulates them via an accumulation register or buffer (e.g., accumulation buffer in TPU [17]).

When multicasting or reduction temporally occurs, it provides temporal reuse opportunity, and vice versa.

We provide a mapping analysis example in Figure 2.4 for deep dive into those two behaviors. Figure 2.4 (a) shows an example CONV2D operation and Figure 2.4 (b) illustrates an example row-stationary [16] style mapping for this operation, and Figure 2.4 (c) shows corresponding data and computation assignment on each PEs in the example mapping. Figure 2.4 (e) shows all the data points accessed by each PE (y-axis) across time steps (x-axis). We can observe *replications of data points*. Such replication of accessed data points indicates data reuse opportunities revealed by a mapping. We highlight and label some of the temporal and spatial reuse opportunities in Figure 2.4 (e). These refer to the replicated accessed data points across time and space (PE), respectively, in the data

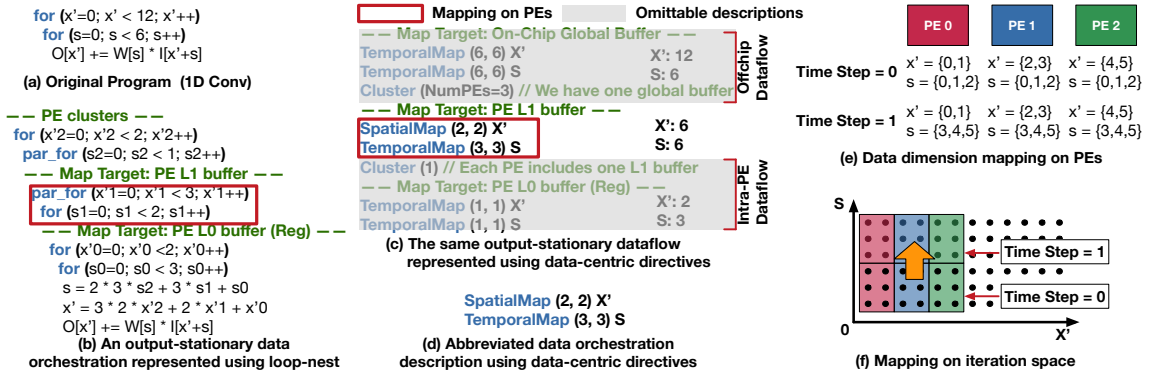


Figure 3.1: An example 1D convolution and an example output-stationary dataflow on the convolution. We represent the dataflow (b) in loop nest and (c) data-centric directives. In (c), gray boxes represent omittable descriptions, which can be inferred (upper gray box) or do not affect the data reuse over PEs (lower gray box). (d) shows an abbreviated form of the dataflow description in data-centric directives. (e) and (f) show resulting mapping on PEs and iteration space, whose dots correspond to computation (or, partial sums).

usage timeline in Figure 2.4 (e). For input and filter values, the temporal reuse can be seen as multicasting into time dimensions, and spatial reuse can be seen as multicasting into spatial dimensions (i.e., PEs). For partial sums for output data, because of the nature of accumulation, they are reduced when they are reused across time or space. Spatial reuse opportunity can be further classified as purely spatial (multicasting in Figure 2.4 (e)) or spatio-temporal (store-and-forward in Figure 2.4 (e)). With proper hardware support, we can leverage such data reuse opportunities, and the resulting data reuse pattern of the example is as given in Figure 2.4 (d).

3.1.3 Existing Representations of Mapping

To convey the scheduling decisions of a particular architecture, dataflows have been expressed as *loop nests*, a syntax that resembles a simple imperative programming language with explicit parallelism, as presented in Eyeriss v2 [27]. We term the loop nest notation a *compute-centric* representation since the data movement is implicit from the loop order and the explicit parallelism specified by the programmer. The loop order dictates the schedule (or, ordering) of computations, the explicit annotation of loops with `parallel-for` captures parallelism, and the combination of loop ordering, tiling, and parallelism enables data reuse. Therefore, architects started to explore optimized loop nests encompassing all of the

three aspects; loop order, parallelism, and tiling. For example, Eyeriss v2 [27] describes its dataflow in a 22-dimensional loop nest.

Compute-centric representations including the polyhedral model have been actively explored. They optimize loops for the parallelism and the locality via a series of loop transformations [89, 90, 91, 92, 93, 94, 95]. Such works target traditional multi-core CPU-based systems and provide sufficiently accurate cost estimations for guiding compilers. However, since they do not precisely model data reuse, estimating throughput and energy efficiency of a DNN accelerator, whose prime optimization goal is maximizing data reuse, is challenging for those works.

Bao et al. [96] developed an analytical model to accurately estimate cache behavior (thereby computing reuses) for a class of affine programs that can be precisely analyzed by a polyhedral model at compile time. However, they use heavyweight linear-algebra frameworks within the polyhedral model to compute reuse, thereby making it impractical to use these techniques on real large applications. Also, it is very challenging for the polyhedral-based frameworks to compute reuse arising from array subscripts involving non-affine expressions or complex subscripts, such as modulus operations which are common in strided convolutions.

In addition, although there exists a body of past compiler work that performs reuse analysis on sequential programs [89, 90, 91, 92, 93, 94, 95, 96], they lack the ability to analyze loop nests with explicit parallelism, while DNN dataflows often contain multiple levels of parallelism. Also, those past works did not consider spatial reuse (which does not refer to the spatial locality in cache-based architectures but data reuse via wires or across PEs) that leverages multicasting and reduction support of accelerators, which plays a key role in estimating the overall throughput and energy efficiency of spatial DNN accelerators.

Such limitations and challenges motivate us to explore an alternative intermediate representation (IR) of dataflows, a *data-centric* representation where data movement and organization are first-class entities. Since data movement is explicit in the data-centric repre-

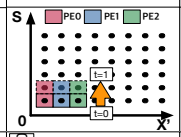
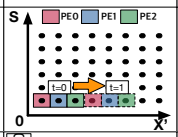
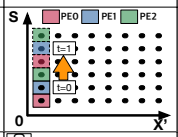
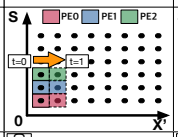
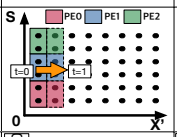
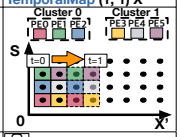
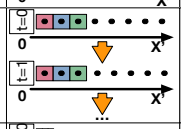
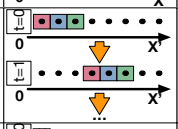
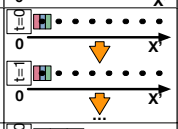
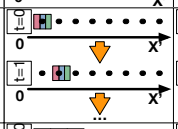
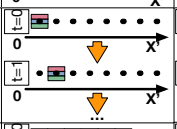
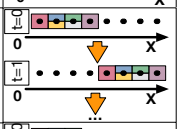
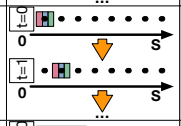
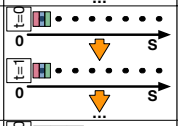
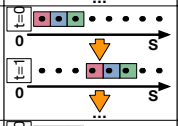
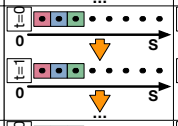
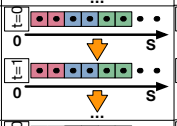
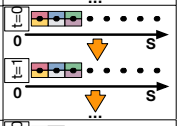
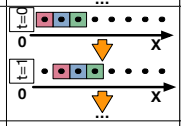
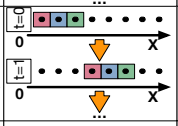
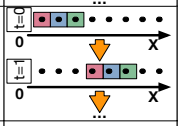
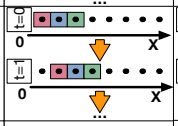
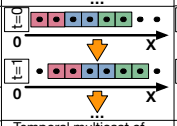
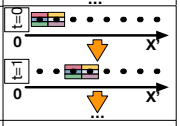
Dataflow ID	A	B	C	D	E	F
Mapping	SpatialMap (1, 1) X' TemporalMap (1, 1) S	TemporalMap (1, 1) S SpatialMap (1, 1) X'	TemporalMap (1, 1) X' SpatialMap (1, 1) S	SpatialMap (1, 1) S TemporalMap (1, 1) X'	SpatialMap (2, 2) S TemporalMap (1, 1) X'	TemporalMap (3, 3) S SpatialMap (1, 1) X' Cluster (3) SpatialMap (1, 1) S TemporalMap (1, 1) X'
Iteration Space (Partial sums)						
Output Featuremap Data Space						
Filter Weight Data Space						
Input Featuremap Data Space						
Temporal Reuse	- Temporal reduction of outputs (Output stationary)	- Temporal multicast of weights (Weight stationary)	- Temporal reduction of outputs (Output stationary)	- Temporal multicast of weights (Weight stationary)	- Temporal multicast of weights (Weight stationary) - Partial temporal multicast of inputs (e.g., X =3 is used by PE1 over t=0 and 1)	- Temporal multicast of weights (Weight stationary)
Spatial Reuse	- Spatial multicast of filter weights	- Spatial multicast of filter weights	- Spatial reduction of outputs	- Spatial reduction of outputs	- Spatial reduction of outputs	- Spatial reduction of outputs
Informal Dataflow Name	Output-Stationary	Weight Stationary	Collaborative Output-Stationary	Collaborative Weight-Stationary	Tiled Collaborative Weight-Stationary	Clustered Tiled Collaborative Weight-Stationary

Figure 3.2: The impact of directive order, spatial/temporal maps, tile sizes, and clustering over 1D convolution presented in Figure 2.10 and Figure 2.11, but with data-centric directives. The first row shows mapping described using the data-centric directives. The second row shows iteration spaces whose points correspond to each partial sum. In row three to five, we show data mapping of each data structure. Finally, we describe temporal and spatial reuse opportunities from each mapping.

sentation, our analytical model becomes simpler and relatively faster as there is no need to leverage heavyweight linear-algebra frameworks to precisely estimate data movement/reuse behavior.

3.2 Describing Mappings

Our data-centric representation consists of data-centric mapping directives and data movement order. The data-centric mapping directives consist of three directives: (1) spatial map, (2) temporal map, and (3) cluster. We discuss how data-centric mapping representation incorporates those four (three directives and data movement order).

3.2.1 Data-Centric Representation

To explicitly describe key aspects of mappings, we propose a data-centric representation that explicitly describes data orchestration (data movement and staging behavior), unlike compute-centric representations implicitly describe them.

The representation is based on four key components we discussed, spatial mapping, temporal mapping, clustering, and data movement.

We the components

1. **Spatial Map(*size*, *offset*) α** specifies a distribution of dimension α (e.g., `R`, `X`) of a data structure across PEs, where `size` refers to the number of indices mapped in the dimension α to each PE, and `offset` describes the shift in the starting indices of α across consecutive PEs.

2. **Temporal Map(*size*, *offset*) α** specifies a distribution of dimension α of a data structure across time steps in a PE, and also the mapped chunk of dimension indices is the same across PEs in a time step. The `size` refers to the number of indices mapped in the dimension α to each PE, and `offset` describes the shift in the starting indices of α across consecutive time steps in a PE.

3. **Cluster(*size*):** Cluster groups *size* number of sub-clusters (e.g., PEs at the lowest level; the base sub-cluster is PE, and we recursively construct gruops of PEs via cluster directives) and changes the mapping target to the constructed groups (or, cluster). Mapping directives under a cluster directive targets sub-clusters within each constructed clusters, and those above the cluster directive targets the constructed clusters.

4. **Data Movement Order:** The sequence of spatial and temporal maps in the dataflow specification dictate the order of data movement, i.e., the change of the data mappings to PEs across time.

We demonstrate reuse opportunities presented by various mappings using the 1D convolution example in Figure 3.1(a). We start by creating a unique mapping for this program by the loop nest representation in Figure 3.1(b), assuming the accelerator has 2-level hierarchy (L0 register at PE + L1 local scratchpad buffer). The two loops enclosed in the red box are indicative of the mapping over the PEs, and their corresponding data-centric representation is in Figure 3.1(c) and (d).

As can be seen from Figure 3.1(e), the data elements corresponding to outputs (dimension `X'`) is spatially distributed across three PEs, i.e., each PE receives different chunks of two output elements. This particular data distribution can be captured with our spatial map directive with size and offset parameters being 2, resulting in `SpatialMap(2,2) X'` where `X'` is the first dimension of output data structure. Also, the data elements corresponding to weights (dimension `S`) is replicated across multiple PEs, i.e., each PE receives a same chunk of three weight elements in the first iteration, and receives different chunk of weight elements in the next iterations. This particular replicated and temporal distribution can be captured with our temporal map directive with size and offset parameter being 3, resulting in `TemporalMap(3,3) S`, where `S` is the first dimension of the weight data structure. Putting it together, spatial map on `X'` followed by a temporal map on `S` captures data mapping and movement behavior across PEs and time corresponding to the two loops in the loop-nest version, and these two directives are enclosed in the red box in Figure 3.1(c). Each data-centric representation is a complete description of a unique mapping.

3.2.2 Understanding the Impact of Mapping

We build six example mappings upon the simple 1D convolution discussed in Figure 3.1 (d) to demonstrate how small changes to a mapping expose various forms of reuse—both spatial and temporal. Figure 3.2 illustrates those six example mappings, which are the same mapping examples in Figure 2.10 and Figure 2.11 but represented in data-centric directives. We modify the data-centric representation (directive order, spatially/temporally mapped

dimensions, mapping size, and PE clustering) and discuss their impact on data reuse.

Directive Order. A change in directive order can result in an entirely different temporal reuse (or, stationary behavior). For example, the sequence of directives in mapping in Figure 3.2(A) indicates that all data indices of S should be explored before working on the next chunk of `X'` indices. This order results in temporally reusing values of data corresponding to `X'` indices (i.e., partial sums) for all indices of `S`. Therefore, this mapping is informally referred to as output-stationary and partitioned across multiple outputs in parallel. Figure 3.2(B) shows the impact of interchanging the order of directives. This results in a weight-stationary mapping, because PEs can temporally reuse weight values corresponding to `S` indices, for all indices of `X'` before going to next chunk of `S` indices. Similarly, Figure 3.2(C) and (D) shows the spatial distribution on `S` instead of `X'`, and also the impact of data movement order on temporal reuse leading to different mapping variations. This indicates why the informal mapping name should not be taken as a complete and precise specification of its behavior.

Spatially and Temporally Mapped Dimensions. In Figure 3.2(A) the directive `SpatialMap(1,1) X'` (where `X'` refers to the first dimension of the output data structure), spatially distributes indices of the `X'` dimension with a chunk size of one (the `size` parameter) across PEs with an offset of one (the `offset` parameter). This means that each PE works on a different column of the output data space. If the number of PEs is not sufficient to cover all indices of the dimension mapped, then the mapping is folded over time across the same set of PEs. Also, if `offset` value is smaller than `size` value, then there will be an overlap of indices across consecutive PEs, and this is useful in describing mappings on input activation dimensions X and Y because their iteration space is skewed.

Similarly, `TemporalMap(1,1) S` (where `S` refers to the first dimension of filter weight data structure), distributes indices of the `S` dimension with a chunk size of one across time steps with an offset of one. This means that each PE works on the same column of the weight data space. Since all PEs get the same data indices corresponding to a temporally

mapped dimension, this creates an opportunity for *spatial reuse*, i.e., multicasting the same data values across PEs in a time step.

Mapping Size. In all of the mappings from Figure 3.2A-D, the mapping sizes (first argument) of weights and outputs are one – resulting in full temporal reuse of weights but no temporal reuse of outputs (e.g., mapping B and D) or vice versa (e.g., mapping A and C). There is no temporal reuse of inputs in any mapping. Increasing the map size of the spatial or temporal maps can help in presenting opportunities for partial temporal reuse, which can capture convolutional reuse of inputs in CNN layers. For example, the spatial map corresponding to the `S` dimension in Figure 3.2(E) helps in exploiting the partial temporal reuse of input data across time steps.

PE Clustering for Multi-dimensional Spatial Distributions. As can be seen in Figure 3.2(A-E), data mappings related to a map in the outer position get updated after a full exploration of a map in the inner position. This inherent assumption can limit certain mapping behaviors where one might be interested in simultaneously exploiting spatial distribution of more than one data dimensions.

To address this, we introduce another directive called *Cluster* as a mean to support the simultaneous spatial distribution of multiple data dimensions. The cluster directive logically groups multiple PEs or nested sub-clusters (when a mapping has multiple cluster directives) of `size` parameter. For example, CLUSTER (3) in Figure 3.2(F) arranges available PEs into groups of three, resulting in two clusters of three PEs.

All the mapping directives specified above a CLUSTER directive perform the mapping across logical clusters created by the CLUSTER directive. All the mapping directives specified below a CLUSTER directive perform the mapping across PEs or lower level logical clusters inside a logical cluster created by the CLUSTER directive. That is, all the mapping directives above a CLUSTER directive see logical clusters while those below the CLUSTER directive see *inside* of each logical cluster. With this mechanism, one can specify complex dataflows with multiple parallelization dimensions represented by multiple SPATIALMAP

Table 3.1: Reuse opportunities based on spatially-mapped dimensions in combination with innermost temporally-mapped dimensions. Filters (F), Inputs (I), and Outputs (O) are considered separately. For brevity, X/Y should be interpreted as X'/Y' as appropriate.

Spatial							Temporal						
Mapped Dim.	Coupling			Reuse Opportunity			Innermost Mapped Dim.	Coupling			Reuse Opportunity		
	F	I	O	F	I	O		F	I	O	F	I	O
K	✓		✓		Multicast		C	✓	✓				Reduction
							R/S	✓		✓		Multicast	
							X/Y		✓	✓	Multicast		
C	✓	✓				Reduction	K	✓		✓		Multicast	
							R/S	✓		✓		Multicast	
							X/Y		✓	✓	Multicast		
R/S	✓		✓		Multicast		K	✓		✓		Multicast	
							C	✓	✓				Reduction
							X/Y		✓	✓	Multicast		
X/Y		✓	✓	Multicast			K	✓		✓		Multicast	
							C	✓	✓				Reduction
							R/S	✓		✓		Multicast	

directives (one in each cluster level). An example of this can be seen in Figure 3.2(F), where the `X'` dimension is spatially distributed across clusters, and the `S` dimension is spatially distributed within the cluster. The cluster directives enable us to represent existing real-world accelerator mappings, such as Eyeriss [16] since it involves the spatial distribution of R and Y dimensions simultaneously, and also NVDLA [49] which involves the spatial distribution of K and C dimensions. Another advantage of the cluster directive is that its notion of grouping multiple PEs can represent coarse-grained PEs in accelerators, such as SIMD units [50] and matrix tensor accelerators like GPU Tensor Cores.

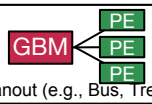
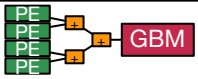
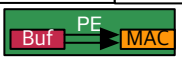
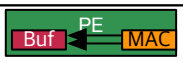
In summary, we discussed five transformations that capture all possible aspects of mappings: scheduling, tiling, and mapping. As shown in Figure 3.2 the data-centric directives can concisely represent all of those aspects. We envision that the data-centric representation could be either auto-generated from a loop nest version of the mapping (with affine constraints), or manually written.

3.2.3 Hardware Implications of Reuse

As we discussed above, various data reuse opportunities appear based on the mapping. We discuss various hardware implementation choices for supporting a wide range of data-reuse across space, time, and space-time.

Table 3.1 summarizes how such opportunities appear in the relationship of spatially mapped dimension within a cluster (Map column) and inner-most temporally mapped

Table 3.2: Hardware Implementation Choices for supporting spatial and temporal reuse. Note - by *temporal multicast*, we refer to *stationary* buffers from which the same data is read over time.

Reuse Type	Communication Type	HW Implementation Choices	
Spatial	Multicast	 Fanout (e.g., Bus, Tree)	 Store-and-Fwd (e.g., Systolic Array)
	Reduction	 Fanin (e.g., Reduction Tree)	 Reduce-and-Fwd (e.g., Systolic Array)
Temporal	Multicast	 Multiple reads from a buffer	
	Reduction	 Multiple read-modify-write to a buffer	

dimension (InnerMap column). For example, if output channels (K) are spatially mapped, a decoupled data structure, input feature map, does not change over space. That is, all the PEs receive the same input feature map, which implies a full spatial reuse opportunity (broadcast). In the same example, when the inner-most temporally mapped dimension is the input channels (C), the input channel changes every iteration, which provides temporal reduction opportunities of outputs.

Although a mapping provides temporal or spatial data reuse opportunities, appropriate hardware support is required to actually exploit these phenomena. Table 3.2 summarizes four reuse categories and corresponding hardware implementation to support them. As the table shows, reuse can be either spatial or temporal. Based on the data structure, the communication type can be either multicast (input tensors) or reduction (output tensors). Multicast is a communication type that delivers the same data to multiple targets over space (different PEs at the same time) or time (the same PE in different time). Therefore, multicast is one to many communication type, which requires either a fan-out network-on-chip structure such as bus or tree, or a "stationary" buffer to hold the data and deliver it to the future. In contrast, the reduction is many to one communication type, which applies to partial sums to generate final outputs. The reduction also can be either spatial or temporal. Example hardware to support spatial reduction is a reduction tree or reduce-and-forward

chain such as systolic arrays. Temporal reduction can be supported by a read-modify-write buffer.

In summary, different mappings (expressed via our directives) expose different forms of reuse: spatial and temporal, both for multicasts and reductions, which in turn can have multiple hardware implementations. Reasoning about mappings in this structured manner exposes new insights and potential microarchitectural solutions. The discussion so far focused on a simple 1D convolution, which itself exposed many possible mappings and reuse opportunities. We extend this to a full convolution loop and analyze reuse opportunities within a specific mapping.

3.2.4 Extended Example: Row-stationary Mapping

Figure 3.3 presents detailed mapping and reuse patterns across two unit time steps of an example row-stationary mapping [16] over a six-PE accelerator. The accelerator has two PE clusters with three PEs in each cluster. Figure 3.3 (a) is the loop nest representation of a row-stationary mapping. Figure 3.3 (b) visualized how the accessed data change across space (PE clusters) and time Figure 3.3 (c) shows the actual coordinates of each tensor across two time steps and two clusters (i.e., time and space). Each colored box in Figure 3.3 (c) represents replicated data points, which imply reuse opportunities. Based on the replicated data points, we can infer data reuse over the PE array, as shown in data reuse row in Figure 3.3(d). The mapping in Figure 3.3 (c) shows that the same set of input activation values are replicated across two clusters in a skewed manner within the same time step, which implies spatial reuse opportunities in the diagonal direction of the example PE array. Similarly, Figure 3.3 (c) shows that the same set of weight values are replicated over two time steps within the same PE, which implies temporal reuse opportunities and weight-stationary style mapping in unit time step granularity. Note that the mapping is still row-stationary in a coarse-grained time step although it is weight stationary in unit time steps we define in Figure 3.3 (a). Finally, Figure 3.3 (c) shows the same set of output

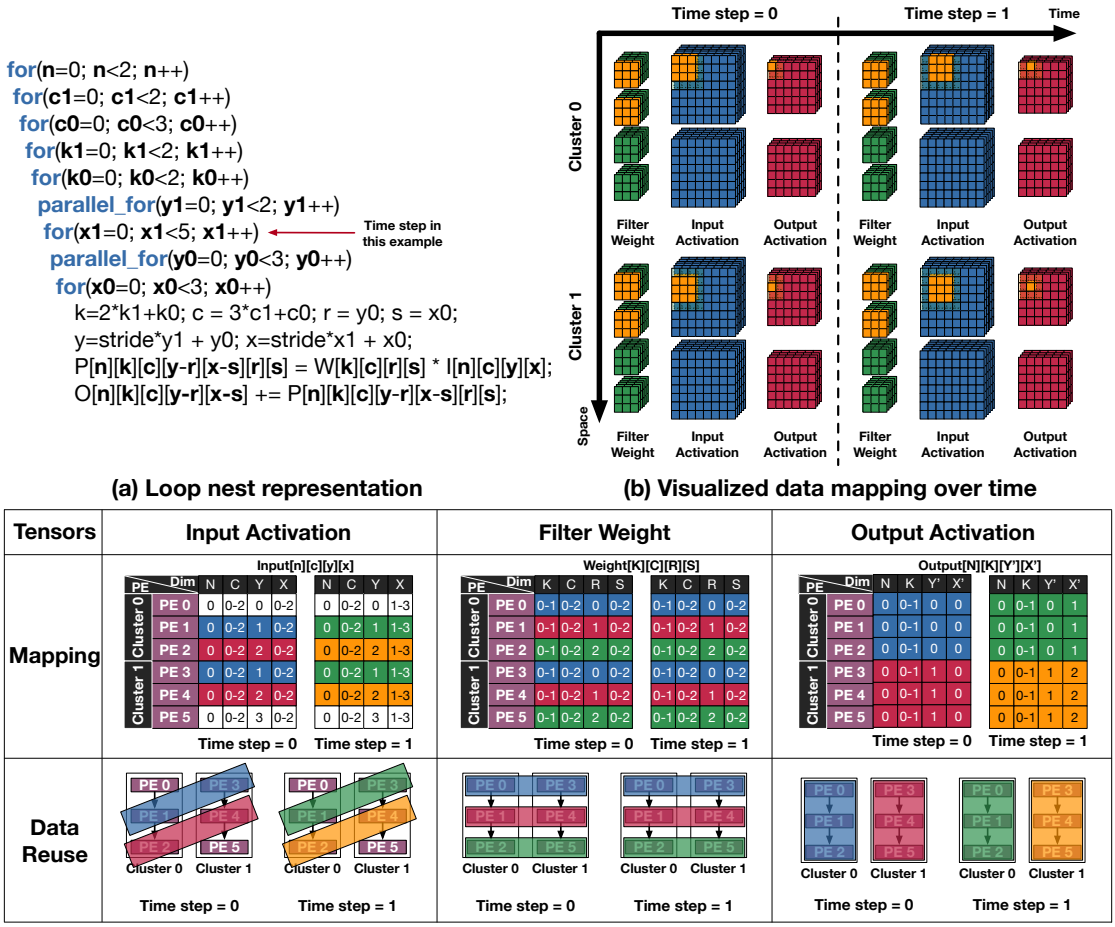
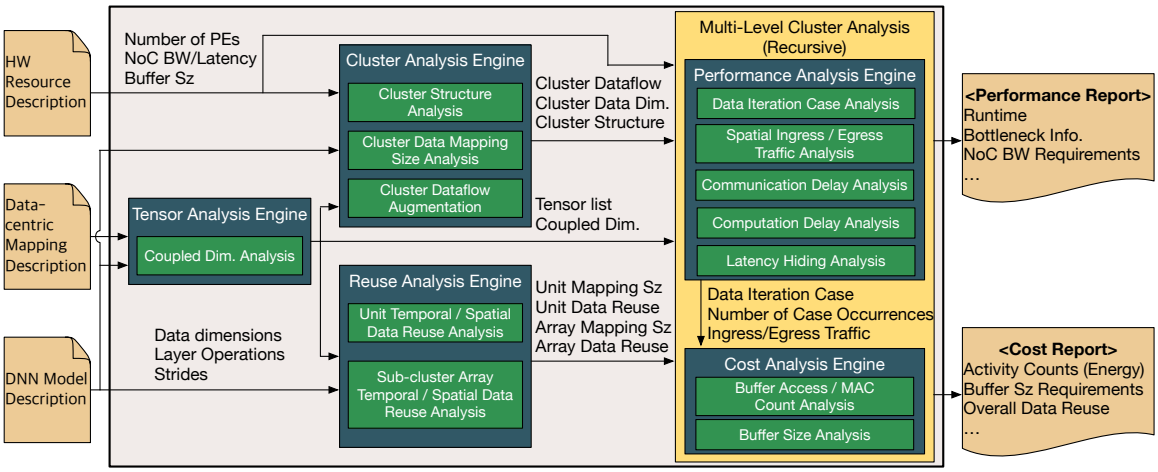


Figure 3.3: An extended example of a row-stationary style dataflow mapped on a six-PE accelerator. We select our own tile sizes for any not specified in the original work [16]. We do not apply additional mapping optimizations to minimize PE under-utilization. Colors represent data replication either across time or space (PEs). Directives with asterisks indicate fully unrolled directives that cover entire data dimension with one mapping.

activation over PEs in each PE cluster, which means that all the PEs in each cluster cooperate to generate a set of output activation data. That is, each PE in a PE cluster generates different partial sums for the same output activation, and they need to be accumulated across PEs in each PE cluster to generate final output activation values.

Based on the example analysis in Figure 3.3, we observe that the data reuse pattern exactly matches the original work [16]: reuse in the horizontal direction for filter weights and vertical for outputs (partial sum accumulation), and reuse in the diagonal direction for input activations.

In summary, reuse opportunities are based on the replicated data across time or space



(a) MAESTRO Analysis Framework

Engine	Analysis Description
Tensor Analysis	- Extract tensor names, tensor types (input/output), coupled dimensions, and size of each coupled dimenisons
Cluster Analysis	- Extract dataflow directives for each cluster level - Extract data dimension sizes for each cluster level - Infer missing directives, apply stride, and so on - Analyze the number of sub-clusters at each cluster level
Reuse Analysis	- Analyze unit mapping size and the amount of reuse across unit steps for every possible combination of data iteration position cases. (***computes temporal reuse***). - Consolidate unit step reuse analysis results to compute reuse over entire sub-cluster array for each cluster level. (***computes spatial reuse***).
Performance Analysis	- Identifies all the possible data iteration cases and the number of occurrences of the cases - Computes ingress/egress traffic considering data reuse - Computes communication (ingress/egress) and computation delay - Considering latency hiding, determine outstanding delay
Cost Analysis	- For each data iteration case, computes the number of buffer accesses and MAC operations - Identify buffer size requirements based on worst-case analysis

(b) Analysis in Each Engine

Figure 3.4: An overview of MAESTRO's analysis framework. For simplicity, we omit components other than analysis engines.

(PEs), which implies temporal and spatial reuse opportunities, respectively. The examples in this section demonstrate the need for a fast, accurate quantitative methodology to compute reuse for complex mappings.

3.3 Quantitative Mapping Analysis

In this section, we present our approach to quantitatively estimating runtime and energy efficiency of mappings on a target DNN model and hardware configuration. Based on the approach, we implement an analysis framework, MAESTRO, which consists of five engines: tensor, cluster, reuse, performance analysis, and cost analysis. Figure 3.4 provides

```
Inputs: A tensor information table (tensor_tbl), a cluster information table (cluster_info_tbl), a mapping information table
with mapping and reuse size for all the possible data iteration position cases for each cluster level (mapping_info_tbl), an
abstract hardware model of the target DNN accelerator(hw_model).
Output: Statistics of the target DNN, accelerator, and dataflow (stats)
Object: To compute the mapping size and the amount of data reuse for all the possible data iteration position cases.
Procedure PerformanceAndCostAnalysisEngine:
  Initialize(stats);
  /* Extracts the cross product of all the possible data iteration cases
     (Init, Steady, and Edge) of each data dimension */
  iteration_cases = ExtractDataIterationCases(tensor_tbl , cluster_info_tbl);
  for each iter_case  in iteration_cases
    num_case_occurrences = GetNumCaseOccurrences(iter_case, tensor_tbl, cluster_info_tbl);
    /* Considering iteration case, compute the number of partial sums for each PE */
    num_psums = GetNumPSums(iter_case, cluster_info_tbl , mapping_info_tbl);
    /* Considering reuse and iteration case, compute the amount of new input tensor data to be fetched from a buffer in
upper cluster levels */
    cluster_ingress_traffic = GetNumNewInputDataPoints(iter_case, tensor_tbl , cluster_info_tbl , mapping_info_tbl);
    /* Considering reuse and iteration case, compute the amount of output tensor  data to be committed to a buffer in
upper cluster levels */
    cluster_egress_traffic = GetNumOutputs(iter_case, tensor_tbl , cluster_info_tbl , mapping_info_tbl);
    //// Core cost analysis ////
    for each tensor in tensor_tbl
      stats.upstream_buffer_read[tensor] += cluster_ingress_traffic[tensor];
      stats.downstream_buffer_write[tensor] += cluster_ingress_traffic[tensor];
      stats.upstream_buffer_write[tensor] += cluster_egress_traffic[tensor];
      stats.downstream_buffer_read[tensor] += num_psums;
      stats.upstream_buffer_size_req[tensor] = 2*Max(stats.upstream_buffer_size_req[tensor],
                                                     cluster_ingress_traffic[tensor],
                                                     cluster_egress_traffic[tensor]);
      stats.downstream_buffer_size_req[tensor] =
                                          2*Max(stats.downstream_buffer_size_req[tensor],
                                                num_psums, cluster_egress_traffic[tensor]);
    end
    //// Core performance analysis ////
    ingress_delay = GetDelay(cluster_ingress_traffic, hw_model);
    egress_delay = GetDelay(cluster_output_traffic, hw_model);
    compute_delay = GetComputeDelay(num_psums, hw_model);
    compute_delay += GetPSumFwdDelay(iter_case, tensor_tbl, cluster_info_tbl, mapping_info_tbl);
    /* Considers double-buffering; treats the initialization case as an exeception */
    if IsFullInit(iter_case) then outstanding_delay = ingress_delay + compute_delay + egress_delay;
    else
      outstanding_delay = Max(ingress_delay, egress_delay, compute_delay);
    end
    stats.run_time  += outstanding_delay * num_case_occurrences;
    stats.num_macs += num_psums * num_active_clusters * num_case_occurrences;
  end
  Return(stats);
endprocedure
```

Figure 3.5: A high-level overview of algorithms in performance and cost analysis engines.

a high-level overview of the five engines. In the interest of space, we only discuss high-level algorithms without edge case handling, multiple layers, and multiple cluster levels. For details, we present them in our open-source repository [97].

3.3.1 Preliminary Engines

Tensor Analysis. As described in Figure 3.4, the tensor analysis engine identifies dimension coupling for each tensor based on specified layer operations. For example, in depth-wise convolutions, output activation is not coupled with the output-channel dimension but coupled with the input channel dimension. Note that depth-wise convolution can be understood either in this manner or by eliminating input channel dimension (C). We select this convention because it aligns with MAESTRO's input-centric cost model. MAESTRO allows users to specify tensors with arbitrary dimension coupling, and such coupling relationship is input to

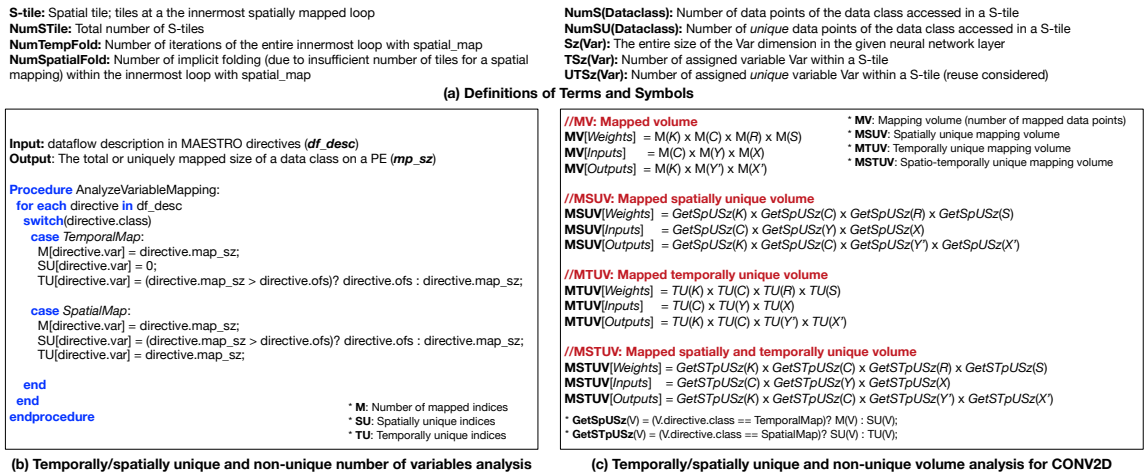


Figure 3.6: A high-level description of preliminary reuse analysis engine. The analysis results are combined with iteration status (i.e., the location of data tile) information to compute exact data reuse.

the rest of engines, which provides generality to MAESTRO.

Cluster Analysis. A PE cluster refers to a group of PEs that processes one or more data dimensions in parallel, specified by the CLUSTER directive. Figure 3.4 (b) describes the analysis in Cluster Analysis (CLA) engine. The CLA engine analyzes a given mapping description written in mapping directives to identify the number of sub-clusters, extract cluster directives and data dimensions, and augment the given mapping descriptions for missing directives, stride handling, and so on, for each cluster level.

Reuse Analysis. Figure 3.4 (b) includes a high-level description of analysis in data reuse analysis (RA) engine. RA engine first identifies data reuse for each PE under all the possible iteration status, which indicates the position of the data tile (initial, steady, and edge). Preliminary data reuse analysis engine analytically computes such information, where we describe the equations in Figure 3.6. Based on the information, RA engine identifies the amount of temporal and spatial reuse across adjacent time steps, which is the data iteration corresponding to the inner-most non-temporally/spatially unrolled mapping directive.

3.3.2 Performance Analysis

Figure 3.4 (a) presents a high-level overview of the performance and cost analysis engine, and Figure 3.5 shows high-level algorithm of the performance analysis (PA) engine. Utilizing

the reuse information computed in the RA engine, PA engine computes the runtime for all the possible cases based on the data dimension and mapping. The computed runtime is multiplied with the number of each case's occurrences and accumulated to compute the total runtime. The runtime of a DNN accelerator consists of communication delay (L2 to L1, L1 to L2, local forwarding) and computation delay in each PE, which are directly related to the accelerator's hardware parameters. PA engine considers double buffering when it computes the outstanding delay (the worst case delay of communication/computation delay) that directly contributes to the runtime.

To estimate communication delays, MAESTRO relies on its analytical network-on-chip (NoC) model based on a pipe model similar to other analytic models [18]. The pipe model utilizes two parameters, the pipe width (bandwidth) and length (average delay), to estimate the communication delay via NoC. The model incorporates a pipelining effect as many packet-switching NoCs have similar behavior. Various combinations of the bandwidth and average delay enables to model NoC structures with reasonable accuracy. For example, Eyeriss [16] has a two-level hierarchical bus with dedicated channels for input, weight, and output tensors. Therefore, a bandwidth of 3X properly models the top level NoC. The average latency depends on implementation details; users should choose an appropriate value considering implementation details (e.g., the use of ingress/egress buffers, which adds one cycle delay each). For more complicated NoC architectures, users should select bisection bandwidth and average latency considering uniform communication to all the PEs from a global buffer. For example, a $N \times N$ 2D mesh network with the injection point at one of the corners, the bisection bandwidth is N, and the average latency is N. Assuming that the user has access to the NoC implementation information, the NoC model is precise when the NoC is a bus or a crossbar.

3.3.3 Cost Analysis

Figure 3.5 describes how the cost analysis (CA) engine computes the number of buffer accesses and estimates the buffer size requirements for each tensor, considering data reuse computed in the RA engine and data iteration cases. Utilizing the access counts and the number of MAC operation information, MAESTRO computes the energy cost. MAESTRO includes an energy model based on those activity counts and Cacti [98] simulation, which can be replaced by any other energy model based on such activity counts (e.g., Accelergy [99]).

3.3.4 Complex Mapping Analysis

Multi-cluster Analysis. Multi-cluster cases can be split into single-cluster cases with the data dimension size set as the mapping size of the corresponding mapping directive in the upper cluster. The outstanding delay of a cluster level becomes the computation delay of the next cluster level above. To handle various edge cases that affects all the lower cluster levels, MAESTRO recursively performs performance and cost analysis, as illustrated in Figure 3.4. In the recursive analysis, the base case is the inner-most cluster whose sub-clusters are actual PEs. Although MAESTRO performs recursion, the complexity is not high because the number of PE cluster levels are typically two or three. Note that each of the edge cases at each cluster level also needs to be recursively processed. However, in most cases, we observe the number of edge cases across cluster levels is less than 20, which is still in a tractable scale.

Other DNNs. Although we used dense convolution as examples for simplicity, MAESTRO can model a variety of layers (LSTM hidden layer, pooling, fully-connected, transposed convolution, and so on) based on the generality of the data-centric approach. Our data-centric approach supports all the operations represented as the loop nest with two input tensors and one output tensor wherein all the tensor indices are coupled in only one or two data dimensions in affine functions.

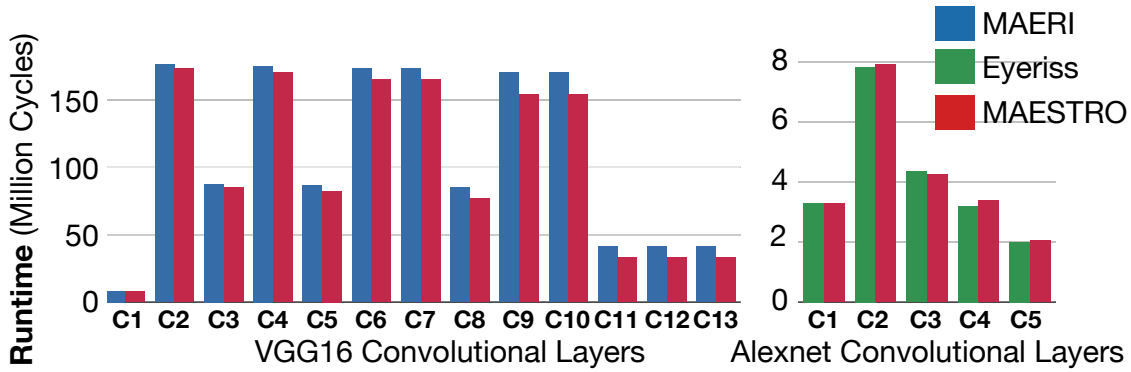


Figure 3.7: Runtime model validation against MAERI [22] RTL simulation with 64 PEs and Eyeriss [63] runtime reported in the paper with 168 PEs.

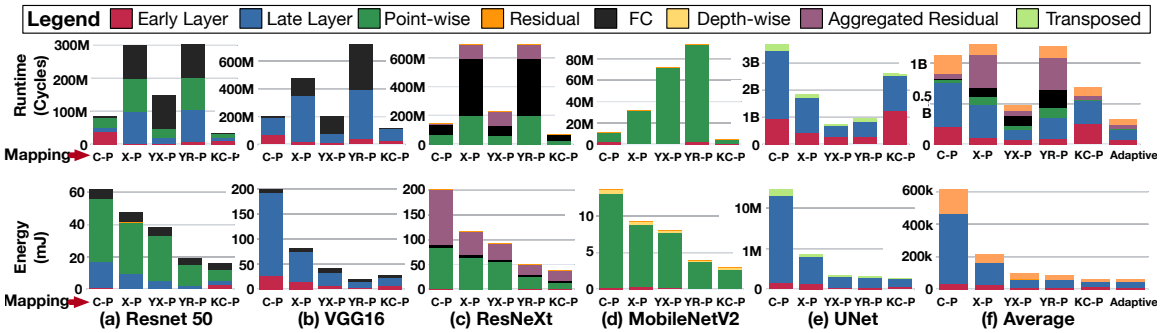


Figure 3.8: Plots in top and bottom rows present runtime and energy estimation of five dataflows listed in the table, respectively. We apply 256 PEs and 32GBps NoC bandwidth. We evaluate all the dataflows using five different DNN model; Resnet50 [14], VGG16 [2], ResNeXt50 [6], MobileNetV2 [5], and UNet [100]. The final column (f) presents the average results across models for each DNN operator type listed in Table 3.4 and the adaptive dataflow case.

3.3.5 Model Validation

We validated MAESTRO's performance model against RTL simulations of two accelerators - MAERI [22] and Eyeriss [63] when running VGG16 and AlexNet respectively[2]. Figure 3.7 shows that the runtime estimated by MAESTRO are within 3.9% absolute error of the cycle-accurate RTL simulation and reported processing delay [63] on average.

3.4 Case Studies

In Table 3.4, we summarize the features of frequently used DNN operators from state-of-the-art DNN models [14, 6, 4, 5, 100]. Early and late layers refer to layers with high-resolution

[2]MAERI RTL is open-source. For Eyeriss, we use the reported runtime for AlexNet because detailed mapping parameters are described for only AlexNet in the paper.

Table 3.3: Five example dataflows used for the evaluation. For conciseness, we omit redundant directives that are automatically inferred by MAESTRO. YX-P, YR-P, and CK-P dataflows are motivated by Shidiannao [61], Eyeriss [16], and NVDLA [49], respectively. The name of each dataflow is based on spatial dimensions from the upper-most cluster level.

Partitioning Strategy	Dataflow	Characteristics
C-Partitioned (C-P)	**TemporalMap** (1,1) K **TemporalMap** (Sz(R),1) Y **TemporalMap** (Sz(S),1) X **TemporalMap** (Sz(R),Sz(R)) R **TemporalMap** (Sz(S),Sz(S)) S **SpatialMap** (1,1) C	- Large spatial reduction opportunities (Large output activation reuse) - Small input activation/filter reuse - Input channel (C) parallelism - No local reuse
X-Partitioned (X-P)	**TemporalMap** (1,1) K **TemporalMap** (1,1) C **TemporalMap** (Sz(R),Sz(R)) R **TemporalMap** (Sz(S),Sz(S)) S **TemporalMap** (Sz(R),1) Y **SpatialMap** (Sz(S),1) X	- Large temporal reuse of filter - Spatial reuse opportunities (via halo in input activation) - Input column (X) parallelism - Weight-stationary
YX-Partitioned (YX-P)	**TemporalMap** (1,1) K **SpatialMap** (Sz(R),1) Y **TemporalMap** (8+Sz(S)-1,8) X **TemporalMap** (1,1) C **TemporalMap** (Sz(R),Sz(R)) R **TemporalMap** (Sz(S),Sz(S)) S **Cluster**(8) **SpatialMap** (Sz(S),1) X	- Large temporal reuse of filter - Better spatial reuse opportunities over X-P (via 2D halo in input activation) - 2D activation (X and Y) parallelism - Output-stationary - Motivated by Shi-diannao [14]
YR-Partitioned (YR-P)	**TemporalMap** (2,2) C **TemporalMap** (2,2) K **SpatialMap** (Sz(R),1) Y **TemporalMap** (Sz(S),1) X **TemporalMap** (Sz(R),Sz(R)) R **TemporalMap** (Sz(S),Sz(S)) S **Cluster**(Sz(R)) **SpatialMap** (1,1) Y **SpatialMap** (1,1) R	- Large temporal reuse of input activation and filter - Spatial reduction opportunities (spatial reuse of output activations) - Activation row (Y) and filter column (S) parallelism - Row-stationary - Motivated by Eyeriss [10]
KC-Partitioned (KC-P)	**SpatialMap** (1,1) K **TemporalMap** (64,64) C **TemporalMap** (Sz(R),Sz(R)) R **TemporalMap** (Sz(S),Sz(S)) S **TemporalMap** (Sz(R),1) Y **TemporalMap** (Sz(S),1) X **Cluster**(64) **SpatialMap** (1,1) C	- Spatial reuse of input activation - Large spatial reduction factor (64-way) over input channel (C) - Input/output channel (C and K) parallelism - Weight-stationary - Motivated by NVDLA [1]

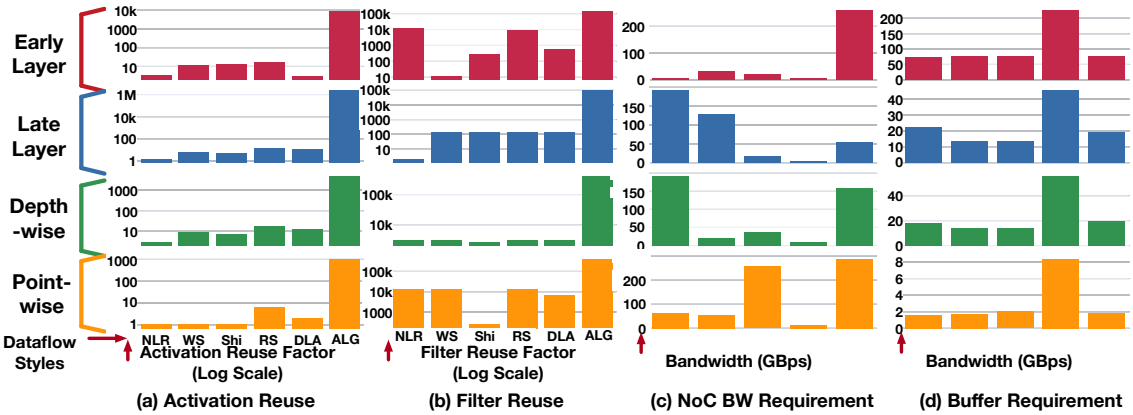


Figure 3.9: Reuse and NoC bandwidth requirements of dataflows in Table 3.3 with 256 PEs for four common DNN operators from Table 3.4. We select representative operators from state-of-the-art DNN models (Early layer: CONV1 in Resnet50 [14], late layer: CONV13 in VGG16 [2], depth-wise convolution (DWCONV): DWCONV of CONV2 in ResNeXt50 [6], point-wise convolution: first conv of bottleneck1 in MobilenetV2 [5] C, X, YX, YR, and KC refers to C-P, X-P, YX-P, YR-P, and KC-P dataflows. A refers to algorithmic maximum reuse.).

activation with shallow channels and vice versa, respectively. We label them as early and late layers because such layers appear early and late in classification networks [14, 6, 5, 2]. We compare the number of input channels and the input activation height to identify them[3].

With MAESTRO, we perform deeper case studies about the costs and benefits of various dataflows when they are applied to different DNN operations listed in Table 3.4. We evaluate five distinct dataflow styles listed in Table 3.3 in Section 3.4.1 and the preference of each dataflow to different DNN operators. For energy estimation, we multiply activity counts with base energy values from Cacti [98] simulation (28nm, 2KB L1 scratchpad, and 1MB shared L2 buffer). We also present distinct design space of an early layer (wide and shallow) and a late layer (narrow and deep) to show the dramatically different hardware preference of different DNN operator styles and dataflow in Section 3.4.2.

3.4.1 Case study I: Dataflow Trade-offs

Figure 3.8 shows the DNN-operator granularity estimation of runtime and energy of each dataflow across five state-of-the-art DNN models listed in Section 3.4. Note that this should be considered a comparison of dataflows—not of actual designs, which can

[3]If C > Y, late layer. Else, early layer

Table 3.4: Operators in state-of-the-art DNNs and their features and implication. Bottleneck [14] and depth-wise separable convolution [4] are listed in a fine-grained operators (point-wise convolution, depth-wise convolution, and residual links). Examples are based on notable networks (VGGnet [2] and DCGAN [15]) and state-of-the-art networks (MobileNetV2 [5], ResNet50 [14], ResNeXt50 [6].

DNN Operators	Examples	Characteristics
CONV 2D Early Layers	- VGG16 CONV1-3 - MobileNetV2 CONV1 - UNet Conv1-2	- Large activation height and width - Shallow input/output channels
CONV 2D Late Layers	- VGG16 CONV4-13 - MobileNetV2 CONV2-3 - UNet Conv3-5	- Small activation height and width - Deep input/output channels
Fully-Connected	- VGG16 FC1-3 - ResNet50 FC1000d - ResNeXt50 FC1000d	- GEMM operation
Point-wise Convolution (1x1 CONV2D)	- ResNet50 CONV2-5 (Bottleneck) - MobileNetV2 Bottleneck 1-7	- No parallelism in filter rows and columns - No convolutional reuse opportunities
Depth-wise Convolution	- MobileNetV2 Bottleneck 1-7	- Reduced computation compared to CONV2D - Reduced data reuse opportunities - Higher NoC BW requirements
Residual Links (Skip Connections)	- ResNet50 CONV2-5 (Bottleneck) - MobileNetV2 Bottleneck1-7	- Extra global buffer / DRAM accesses to fetch previous activation or buffer space for entire activation
Aggregated Residual Blocks	- ResNeXt50 CONV2-5	- More data parallelism via branching structure - Concatenation and reduction of activation required
Transposed Convolution	- UNet UpConv 1-4 - DCGAN CONV1-4	- Upscaled output activations - Structured sparsity in output activations

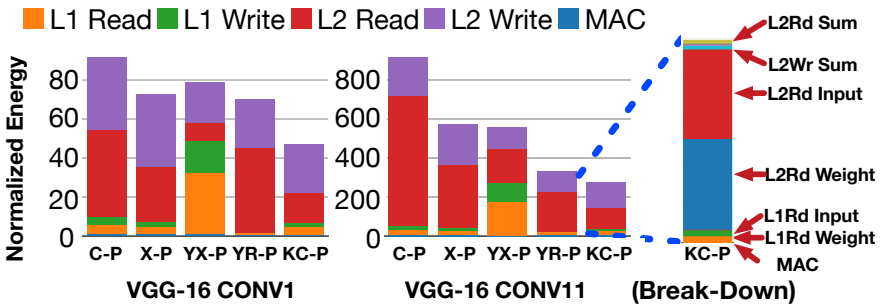
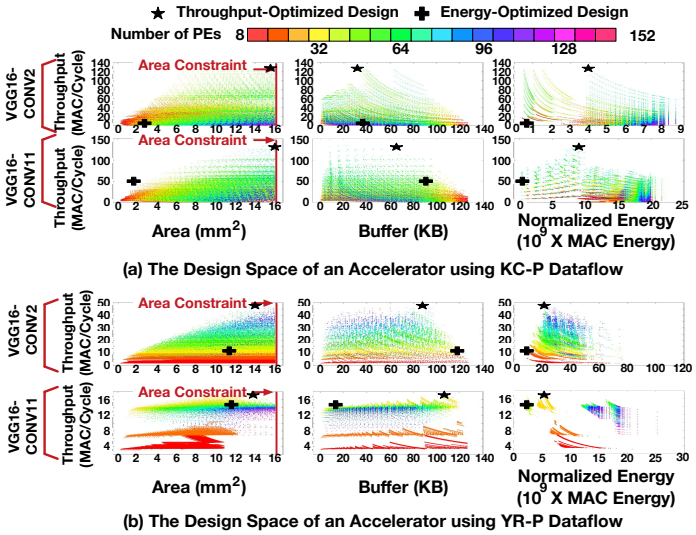


Figure 3.10: The breakdown of energy consumption (MAC and L1/L2 scratchpad access energy) of the dataflows from Table 3.3. The access counts generated by MAESTRO are multiplied by appropriate energy values from Cacti [98]. The values are normalized to the MAC energy of C-P.

contain several low-level implementation differences, e.g., custom implementations of logic/memory blocks, process technology, and so on. We observe that KC-P dataflow style dataflow provides overall low runtime and energy. However, the energy efficiency in VGG16 (Figure 3.8 (b)) is worse than YR-P (Eyeriss [16] style) dataflow, and the runtime is worse than YX-P (Shidiannao [61] style) dataflow in UNet (Figure 3.8 (e)). This is based on the different preference toward dataflow of each DNN operator. YX-P provides short runtime to segmentation networks like UNet, which has wide activation (e.g., 572x572 in the input layer) and recovers the original activation dimension at the end via up-scale convolution (e.g., transposed convolutions). Such a preference to the YX-P style is mainly based on its parallelization strategy: it exploits parallelism over both of row and column dimensions in activation. The energy efficiency of YR-P dataflow in VGG16 is based on its high reuse factor (the number of local accesses per fetch) in early layers, as shown in red bars in Figure 3.9 (a) and (b) (note the log scale). The YR-P dataflow has 5.8$\times$ and 15.17$\times$ higher activation and filter reuse factors, respectively, in early layers. However, in late layers, the reuse factors of YR-P and KC-P dataflow are almost similar (difference $<$ 11%), so the KC-P dataflow provides similar energy efficiency as YR-P in these cases. This can also be observed in the late layer (blue) bars in Figure 3.8 bottom-row plots.

Although the KC-P and YX-P dataflows provide low runtime (Figure 3.8), it comes with high NoC cost, as the high bandwidth requirements shown in Figure 3.9 (c) highlight. Based on the operator type, some dataflows require dramatically higher NoC bandwidth



DSE Setting / Stats	KC-P Dataflow		YR-P Dataflow	
	VGG16 CONV2	VGG16 CONV11	VGG16 CONV2	VGG16 CONV11
Number of Valid Design Points	175,861	260,028	319,268	115,209
Number of Explored Design Points	3,936,256	251,920,384	3,555,328	227,540,992
DSE Execution Time (min)	12.20	18.29	23.68	8.22
DSE Rate (designs/ sec)	5377.40	229561.13	3239.77	461356.43

(c) DSE Statistics

Figure 3.11: The design space of an accelerator with (a) KC-P and (b) YR-P dataflow. We highlight the design space of an early and a late layer to show their significantly different hardware preference. We apply the area and power of Eyeriss [63] as area/power constraints to the DSE.(16mm^2, 450mW [63]). The color of each data point indicates the number of PEs. Design points with fewer PEs can be paired with larger buffer sizes, up to the area budget. We mark the throughput- and energy-optimized designs using a star and cross.

than others. For example, YX-P requires high bandwidth for point-wise convolution as it has no convolutional reuse (i.e., overlapped activation data points among sliding windows) because of its 1x1 kernel while YX-P is optimized to exploit convolutional reuse via spatial reuse.

The diverse preference to dataflows of different DNN operators motivates us to employ optimal dataflow for each DNN operator type. We refer such an approach as adaptive dataflow and present the benefits in Figure 3.8 (f), the average case analysis across entire models in DNN operator granularity. By employing the adaptive approach, we could observe a potential 37% runtime and 10% energy reduction. Such an optimization opportunity can be exploited by flexible accelerators like Flexflow [21] and MAERI [22] or via heterogeneous accelerators that employ multiple sub-accelerators with various dataflow styles in a single DNN accelerator chip.

Table 3.5: The impact of multicasting capability, bandwidth, and buffer size. Design points are from the design space of Figure 3.11 (a) VGG16-CONV2.

Design Point	Num PEs	NoC BW (Data pt/Cycle)	Spatial Reuse Support		Temporal Reuse Support	Throughput (MAC/cycle)	Energy (X MACs)
			Multicast	Reduction	Buffer Size (KB)		
Reference	56	40	Yes	Yes	6.13	48.6	5.26×10^9
Small bandwidth	56	24	Yes	Yes	6.13	34.54	5.26×10^9
No multcast	56	40	No	Yes	2.26	33.39	7.56×10^9
No Sp. reduction	56	40	Yes	No	4.68	32.05	7.77×10^9

3.4.2 Case study II: Hardware Design-Parameters and Implementation Analysis

Using MAESTRO, we implement a hardware design space exploration (DSE) tool that searches four hardware parameters (the number of PEs, L1 buffer size, L2 buffer size, and NoC bandwidth) optimized for either energy efficiency, throughput, or energy-delay-product (EDP) within given hardware area and power constraints. The DSE tool receives the same set of inputs as MAESTRO with hardware area/power constraints and the area/power of building blocks synthesized with the target technology. For the cost of building blocks, we implement float/fixed point multiplier and adder, bus, bus arbiter, and global/local scratchpad in RTL and synthesis them using 28nm technology. For bus and arbiter cost, we fit the costs into a linear and quadratic model using regression because bus cost increases linearly and arbiter cost increases quadratically (e.g., matrix arbiter).

The DSE tool sweeps a target design space specified in the range of each parameter and search granularity. However, it skips design spaces at each iteration of hardware parameters by checking the minimum area and power of all the possible design points from inner loops of hardware parameters. This optimization allows it to skip invalid design points in a various granularity that reduces a large number of futile searches, which led to a large effective DSE rate ranging from 3.3K to 0.46M designs per second, as presented in Figure 3.11 (c). Figure 3.11 (c) shows statistics of four DSE runs explored the design space. We ran DSEs on a machine with i7-8700k CPU and 32GB memory operating Linux Mint 19 OS. We run four sets of the DSE on the machine at the same time, and all of them terminated within 24 minutes, with effective DSE rate of 0.17M designs per second, on average.

Design Space Analysis: Using the DSE tool, we explore the design space of KC-P and

YR-P dataflow accelerators. We set the area and power constraint as 16mm^2 and 450mW, which is the reported chip area and power of Eyeriss [63]. We plot the entire design space we explored in Figure 3.11.

Whether an accelerator can achieve peak throughput depends on not only the number of PEs but also NoC bandwidth. In particular, although an accelerator has sufficient number of PEs to exploit the maximum degree of parallelism a dataflow allows, if the NoC does not provide sufficient bandwidth, the accelerator suffers a communication bottleneck in the NoC. Such design points can be observed in the area-throughput plot in Figure 3.11 (a). YR-P dataflow requires low NoC bandwidth as shown in Figure 3.9 (c) so it does not show the same behavior as KC-P dataflow. However, with more stringent area and power constraints, YR-P dataflow will show the same behavior.

During DSE runs, MAESTRO reports buffer requirements for each dataflow and the DSE tool places the exact amount buffers MAESTRO reported. Contrary to intuition, larger buffer sizes do not always provide high throughput, as shown in buffer-throughput plots in Figure 3.11 (plots in the second column). The optimal points regarding the throughput per buffer size are in the top-left region of the buffer-throughput plots. The existence of such points indicates that the tiling strategy of the dataflow (mapping sizes in our directive representation) significantly affects the efficiency of buffer use.

We also observe the impact of hardware support for each data reuse, discussed in Table 3.2. Table 3.5 shows such design points found in the design space of KC-P dataflow on VGG16-conv2 layer presented in the first row of Figure 3.11 (a). The first design point is the throughput-optimized design represented as a star in the first row of Figure 3.11. When bandwidth gets smaller, the throughput significantly drops, but energy remains similar. However, the lack of spatial multicast or reduction support resulted in approximately 47% energy increase, as the third and fourth design points shows.

We observe that the throughput-optimized designs have a moderate number of PEs and buffer sizes, implying that hardware resources need to be distributed not only to PEs but

```
Network Resnet50 {
    Layer CONV1 {
        Type: CONV
        Stride { X: 2, Y: 2 }
        Dimensions { K: 64, C: 3, R: 7, S: 7, Y:224, X:224 }
        Mapping {
            SpatialMap(1,1) K;
            TemporalMap(64,64) C;
            TemporalMap(Sz(R),Sz(R)) R;
            TemporalMap(Sz(S),Sz(S)) S;
            TemporalMap(Sz(R),1) Y;
            TemporalMap(Sz(S),1) X;
            Cluster(64, P);
            SpatialMap(1,1) C;
            TemporalMap(Sz(R),1) Y;
            TemporalMap(Sz(S),1) X;
            TemporalMap(Sz(R),Sz(R)) R;
            TemporalMap(Sz(S),Sz(S)) S;
        }
    }
    Layer CONV2_1_1 {
        Type: CONV
        Stride { X: 1, Y: 1 }
        Dimensions { K: 64, C: 64, R: 1, S: 1, Y: 56, X: 56 }
        Mapping {
...
```

(a) Example Input File of MAESTRO

Neural Netw	Layer Numb	NumPEs	Runtime (Cycles)	Energy (nJ)	Throughput	Throughput	Area	Power	N
Resnet50	CONV1	16384	18988903	1.70E+09	712.556	418.789	1.38E+11	161924	
Resnet50	CONV2_1_1	16384	41022176	1.06E+11	2044.9	19.2717	1.38E+11	147469	
Resnet50	CONV2_1_2	16384	161344232	5.39E+11	3960.54	7.35468	1.38E+11	147477	
Resnet50	CONV2_1_3	16384	163836419	3.98E+12	2048.04	0.514521	1.38E+11	147517	
Resnet50	CONV2_1_R	16384	802856	1.49E+09	31.9984	21.4756	1.38E+11	154676	
Resnet50	CONV2_2_1	16384	164094176	9.36E+10	2044.83	21.8386	1.38E+11	147469	
Resnet50	CONV2_2_2	16384	161344232	2.09E+12	3960.54	1.89133	1.38E+11	147477	
Resnet50	CONV2_2_3	16384	163836419	3.98E+12	2048.04	0.514521	1.38E+11	147517	
Resnet50	CONV2_2_R	16384	802856	1.49E+09	31.9984	21.4756	1.38E+11	154676	
Resnet50	CONV2_3_1	16384	164094176	9.36E+10	2044.83	21.8386	1.38E+11	147469	
Resnet50	CONV2_3_2	16384	161344232	2.09E+12	3960.54	1.89133	1.38E+11	147477	
Resnet50	CONV2_3_3	16384	163836419	3.98E+12	2048.04	0.514521	1.38E+11	147517	
Resnet50	CONV2_3_R	16384	802856	1.49E+09	31.9984	21.4756	1.38E+11	154676	
Resnet50	CONV3_1_1	16384	3934219	3.08E+09	2063.99	670.872	1.38E+11	147449	
Resnet50	CONV3_1_2	16384	379890	5.32E+10	7763.09	145.834	1.38E+11	147485	
Resnet50	CONV3_1_3	16384	744095	1.91E+10	2054.61	107.804	1.38E+11	147497	
Resnet50	CONV3_1_R	16384	401456	7.45E+08	31.9962	42.9231	1.38E+11	154680	
Resnet50	CONV3_2_1	16384	370713	5.19E+08	2062	3970.38	1.38E+11	147469	
Resnet50	CONV3_2_2	16384	379890	6.04E+09	7763.09	1285.5	1.38E+11	147485	
Resnet50	CONV3_2_3	16384	744095	1.91E+10	2054.61	107.804	1.38E+11	147497	
Resnet50	CONV3_1_R	16384	401456	7.45E+08	31.9962	42.9231	1.38E+11	154680	
Resnet50	CONV3_3_1	16384	370713	5.19E+08	2062	3970.38	1.38E+11	147469	
Resnet50	CONV3_3_2	16384	379890	6.04E+09	7763.09	1285.5	1.38E+11	147485	
Resnet50	CONV3_3_3	16384	744095	1.91E+10	2054.61	107.804	1.38E+11	147497	
Resnet50	CONV3_3_R	16384	401456	7.45E+08	31.9962	42.9231	1.38E+11	154680	
Resnet50	CONV3_4_1	16384	370713	5.19E+08	2062	3970.38	1.38E+11	147469	
Resnet50	CONV3_4_2	16384	379890	6.04E+09	7763.09	1285.5	1.38E+11	147485	
Resnet50	CONV3_4_3	16384	744095	1.91E+10	2054.61	107.804	1.38E+11	147497	
Resnet50	CONV3 4 R	16384	401456	7.45E+08	31.9962	42.9231	1.38E+11	154680	

...

...

(a) Example Output File of MAESTRO

Figure 3.12: Example input and output files of MAESTRO. In the output file, we show limited number of columns in the interest of space.

also to NoC and buffers for high PE utilization. Likewise, we observe that the buffer amount does not directly increase throughput and energy efficiency. These results imply that all the components are intertwined, and they need to be well-balanced to obtain a highly-efficient accelerator.

3.5 MAESTRO Codebase and Availability

The source code of MAESTRO is available as open-source via the following link: https://github.com/georgia-tech-synergy-lab/maestro. General information and documentation is available via the following web site: http://maestro.ece.gatech.edu.

Figure 3.12 (a) and (b) show example input and output files of MAESTRO. The input file is a listing of layers with mapping specified in data-centric directives as shown in Figure 3.12 (a). In addition to the mapping file, a hardware description file is also required, which is a simple listing of hardware parameters (number of PEs, NoC bandwidth, buffer size, etc.) and their values. The output file is a csv file that contains statistics for all the layers listed in the input file. The number of statistics for each layer is 50, which includes estimated runtime, energy, throughput, buffer size requirements, NoC bandwidth requirement, multicasting factor, and so on.

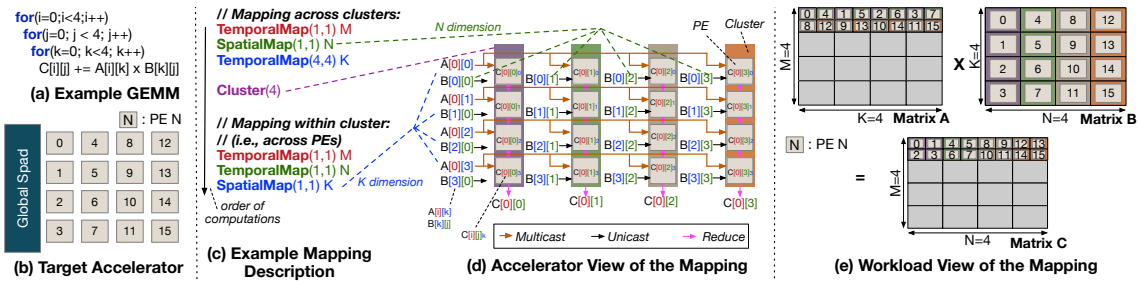


Figure 3.13: A walk-through Example of GEMM mapping.

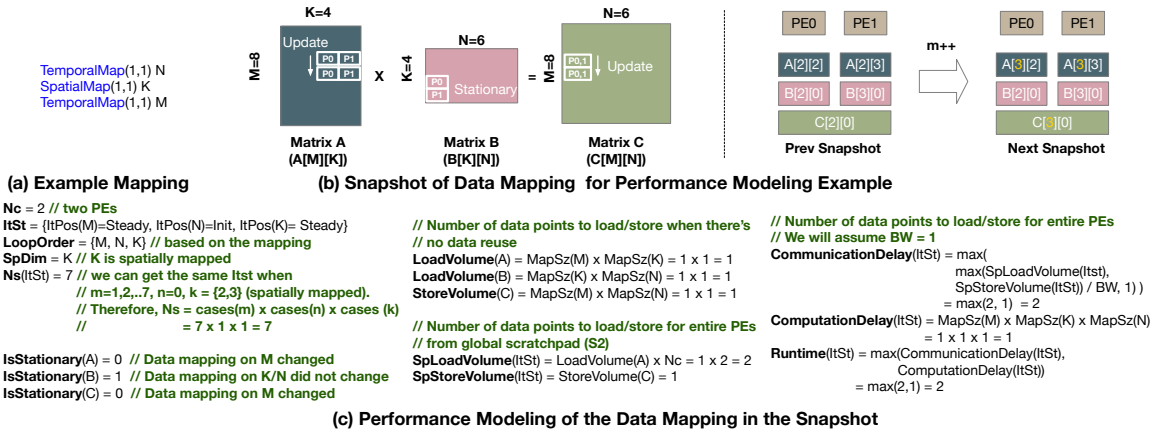


Figure 3.14: An example of cost-modeling within BLAS extension of MAESTRO to compute data reuse and runtime for a snapshot of execution. Nc is number of sub-clusters (i.e., PEs), ItSt is iteration status that indicates the position of the data mapping in process, SpDim is spatially mapped dimension (i.e., parallelized dimension), and Ns is number of the iteration status occurrence.

3.6 BLAS Extension

General matrix multiplication (GEMM) operation is the backbone of basic linear algebra subprograms (BLAS) operations, which are key operations in high-performance computing domain. We first show how mappings on GEMM operation can be represented using data-centric mapping directives and discuss the cost modeling with an example.

3.6.1 Describing GEMM Mappings

Figure 3.13 (a) shows an example GEMM (with M=4, N=4 and K=4) to be mapped on a 16-PE accelerator shown in Figure 3.13 (b). Figure 3.13 (c) shows an example mapping for the example GEMM using the data-centric mapping directives of MAESTRO. We describe three key facets of this mapping, that can be seen in Figure 3.13 (d) and Figure 3.13 (e)

from the perspective of the accelerator and the workload, respectively.

Clustering. In the example mapping, the 16 PEs are divided into four clusters, each with four PEs (specified via the Cluster directive). This allows the mapping to exploit the 2D nature of the physical PE array.

Intra-Cluster Behavior. Mapping directives below the cluster directive specify the intra-cluster mapping. Within each cluster (i.e., column of the accelerator), the K dimension is mapped spatially (specified via the SpatialMap), while the M and N dimensions are mapped temporally (i.e., remain same across all the PEs and only change with time). All directives use a *Size* and *Offset* of 1. The outcome of these directives is to specify that each PE receives *one* unique element from the row and column of the matrices A and B respectively, since M and N stay the same, but K changes in each PE. This can be visualized from Figure 3.13(d). During operation, each PE computes a partial sum and forwards it to its neighbor along the column for accumulation. Each cluster thus computes one element of matrix C.

Inter-Cluster Behavior. Mapping directives above the cluster directive specify the inter-cluster mapping. Across the clusters, the N dimension is mapped spatially, while the M and K dimensions are temporal. This means that the elements of the matrix B (i.e., $K \times N$) gets distributed across the different clusters, while the elements of the matrix A (i.e., $M \times K$) remain the same (and can thus be multicast). Note that the *size* and *offset* field for K is four to specify that each cluster receives four elements from each matrix (which then get distributed among the 4 PEs within the cluster as discussed above). If this field is not set appropriately, it can lead to under-utilization within the cluster.

From Figure 3.13(e), we can observe that each time-step of the mapping computes one row of outputs for the matrix C, and would move on to the next row in the next time-step. If the dimensions of the matrix exceed the dimensions of the physical array, the computation would need to be tiled. The computation order across tiles of the three matrices depends on the order in which the directives are specified (Figure 3.13(b)).

3.6.2 Cost-modeling

Figure 3.14 shows an example of the cost modeling for a GEMM operation. For simplicity, we assume a two PE system in the figure. Figure 3.14(a) shows the mapping description, and Figure 3.14(b) shows a snapshot of the mapping. Note that we use M,K, and N as the dimension identifiers since we target GEMM, not CONV2D. The dimensions of matrix A,B, and C are MxK, KxN, and MxN, as illustrated in Figure 3.14(b). For simplicity, we focus on the cost modeling of a snapshot of the execution, presented in Figure 3.14(b).

We can identify data reuse opportunities. Data operands that do not change over time (*aka stationary [16]*) can stay buffered within the PE buffer. This is the case for *B*[2][0] and *B*[3][0]. *C*[2][0] is mapped on both PE0 and PE1 and can be accumulated via spatial reduction. Across time-steps, both PEs need new elements from the matrix *A* and start working on a new *C* element. Figure 3.14(c) encodes this behavior into a set of equations. Anytime an operand changes (across space or time), a new *volume of data* needs to be fetched depending on the size of the mapped dimensions. Based on the reuse information, the extended MAESTRO computes various statistics such as runtime, buffer access counts, and so on, which can be translated into performance, energy, and HW overhead.

3.7 Summary

Data-centric mapping directives and MAESTRO are motivated by the observation that data orchestration (data movement and staging) dictates the computational performance and energy efficiency of a DNN accelerator. That implies that co-optimizing DNN accelerator microarchitecture and data orchestration via mapping is crucial for maximizing the computational performance and energy efficiency of a DNN accelerator.

In this chapter, we introduced data-centric directives to specify data mappings in a compact form and understand data reuse opportunities via explicit data orchestration information revealed by the data-centric representation.

Based on the explicit information from data-centric mapping representation, we presented an analytical cost model called MAESTRO to estimate execution time, energy efficiency, and the hardware cost of mappings, focusing on data orchestration. We validated our analytical model against the RTL simulation results of MAERI and reported processing time of Eyeriss accelerators and found our model to be highly consistent (96% accuracy, on average), which shows the soundness of the analytic model.

In our cases studies using MAESTRO, we show that diverse preference of DNN layers to mapping and hardware, which motivates flexible mapping accelerator, this thesis mainly discusses. As we discussed in Section 1.1, we propose two approaches - reconfigurable and heterogeneous DNN accelerators - to design flexible mapping accelerators. We first discuss the enabler of reconfigurable accelerators, flexible network-on-chip (NoC) in the following chapter, which maximizes the operational utilization (i.e., $\frac{\text{(The avg. number of active PEs)}}{\text{(The number of PEs with assigned computations)}}$, mainly models PE stalls).

CHAPTER 4

MICROSWITCHES: LIGHT-WEIGHT NETWORK-ON-CHIP DESIGN SPECIALIZED FOR DNN ACCELERATOR TRAFFIC

To design a reconfigurable DNN accelerator, properly supporting varying communication patterns from different mappings is crucial. Considering the goal of accelerators, providing high computational performance (high throughput, low latency, etc.) with lower cost (energy, hardware area, etc.), the requirements on NoC in reconfigurable DNN accelerators are as follows:

- **Capability:** NoCs should provide connectivity for on-chip communication from any mapping to be supported.
- **Bandwidth:** NoCs should provide sufficient bandwidth so that the communication delay does not dominate.
- **Cost-effectiveness:** Hardware area and energy costs need to be light-weighted so that an accelerator can house more compute units.

In this chapter, we analyze DNN accelerator's on-chip traffic patterns and propose a specialized NoC design for the traffic pattterns we analyzed, which satisfies all the tree requirements.

4.1 Traffic in DNN Accelerators

A network-on-chip (NoC) that supports any traffic from mappings to be supported is an enabler for the reconfigurable DNN accelerator because each mapping corresponds to one distinct data orchestration (data movement and staging behavior). However, supporting random traffic is costly, so we choose to specialize NoC architecture for DNN accelerators as they specialize the architecture for DNN computation. For the approach, we first observe

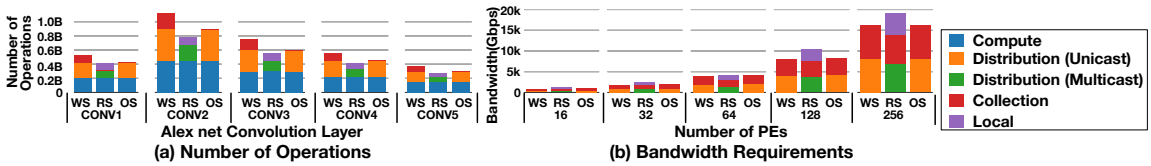


Figure 4.1: (a) Compute (Multiplications and Additions) vs. Communication (distribution/collection/Local) of each Alexnet layer [1] across different CNN implementations: weight stationary (WS), row stationary (RS), and output stationary (OS). (b) Average NoC bandwidth requirement for Alexnet vs. number of PEs

that the traffic in DNN accelerators is not random, but it falls into three classes of flows, distribution, collection, and local forwarding.

- **Distribution:** Distribution is data movement from the global buffer to a PE for operand fetch. distributions can either be unicast or multicast, depending on the dataflow and the mapping of compute on PEs.
- **Collection:** Collection is the traffic flow from PEs to the global buffer for sending computation results in each PE. Collection is usually many-to-one trafffic from PEs to the global buffer.
- **Local Forwarding:** Local forwarding refers to the inter-PE communication traffic, which is a distringuished feature from GPUs. Local forwarding is usally between adjacent PEs, and it could be in the form of unicasts, multicasts, or reductions.

The communication based on those patterns is dominant activity in DNN accelerators, as plotted in Figure 4.1 (a). Figure 4.1 (a) shows the total number of computations and communication flows (distribution/collection/local) when running the convolution layers of AlexNet [1] using mapping styles based on weight-stationary (WS), output-stationary (OS), and row-stationary (RS) dataflows [16]. We can observe that the raw communication to compute ratio is high (larger than 1) regardless of mapping styles, which implies that communication is critical in DNN accelerators to exploit the full compute power from the PE array. That is, to maximize operational utilization [1]

Figure 4.1 (b) translates the raw communication into a bandwidth requirement as a

[1]Unlike mapping utilization, operation utilization refers to the average number of busy compute unit divided by the number of compute unit with assigned operations by a mapping. That is, operational utilization decreases when PE stall occurs because of the delayed operand arrival.

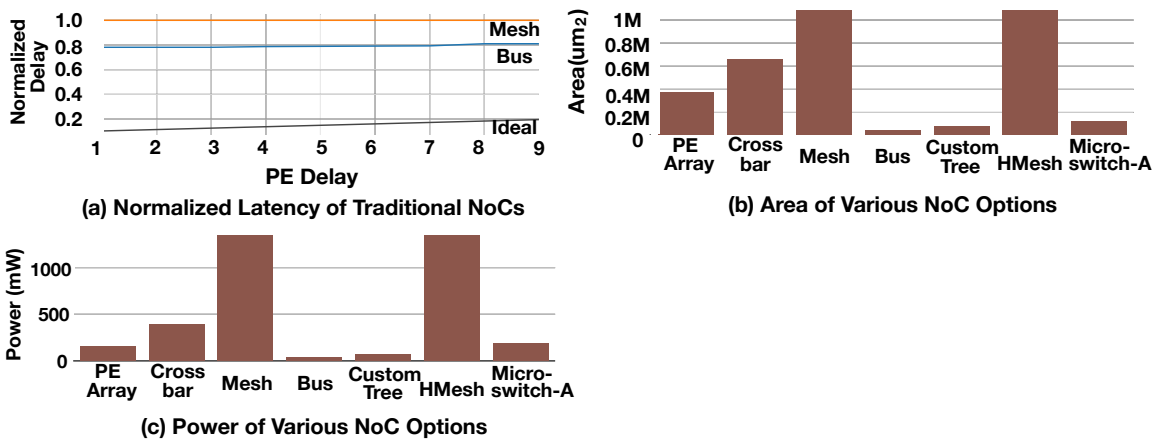


Figure 4.2: Challenges with traditional NoCs for accelerators. (a) Latency of 64-PE WS CNN accelerator with increasing PE delay (b) Area, and (c) Power

function of the number of PEs based on simulation results on Alexnet [1]. For all designs, we observe that distribution bandwidth (unicast for WS and OS, multicasts for RS) is extremely crucial. For WS style mapping, we observe that the bandwidth required for the collection is significant. Finally, we observe that the bandwidth across all traffic flows increases as the number of PEs increases. The analysis suggests that each mapping style requires different amount of bandwidth to NoC, and NoC bandwidth is critical for scalability of the accelerator. Based on the observation, we can conclude that high bandwidth is major requirement for NoCs in DNN accelerators not only for providing scalability but also for supporting various mappings. Can we provide such bandwidth using existing NoC designs used in multicore processors? We explore such options and highlight challenges of employing such NoCs in reconfigurable DNN accelerators.

4.2 Challenges for Traditional NoCs

Traditional NoCs such as buses, meshes, and crossbars are common across multicore processors today. Naturally, they have been integrated into multi-PE DNN accelerators. For example, Eyeriss [63] and DNNWeaver [71] use buses, DianNao [60] and ShiDianNao [61] use meshes, and TrueNorth [101] uses crossbars and meshes in a hierarchical manner. However, such NoCs face major scalability challenges when used inside accelerators. We quantify the challenges and provide intuition about the challenges via two motivational

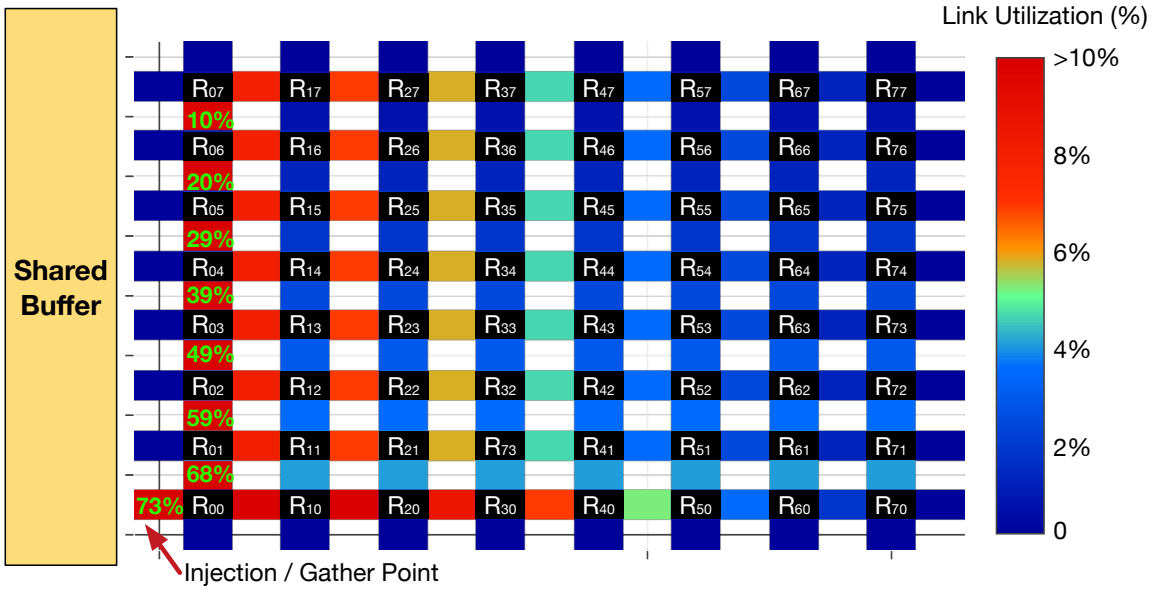


Figure 4.3: Link utilization of 8x8 mesh links running row-stationary [16] style mapping studies.

4.2.1 Motivational Studies

Latency, Area, and Power

We perform a study with various traditional NoCs with a weight-stationary style dataflow. In this study, we assume no storage in PEs, which could mitigate the performance challenges of traditional NoCs at the cost of increased area and power in each PE, to focus on the pure impact of communication. We first quantify the total latency of executing CONV1 layer of Alexnet with 64 PEs using mesh, bus, and an ideal NoC that implements a single-cycle zero-contention network in Figure 4.2 (a). We also analyze area and power of NoCs compared to a PE array with 64 Eyeriss [16][2] PEs in Figure 4.2 (b) and (c).

Performance. In Figure 4.2 (a), We compare the performance of a mesh, and a multi-bus/multi-tree topology against an "ideal" NoC which is a single-cycle zero-contention network. From the results, we make two observations as follows:

- With a 1-cycle PE, we observe that the mesh and a single bus or tree is 10× slower than the ideal. The reason is heavy contention at links near the global buffer. Assuming that

[2]We thank the Eyeriss authors for sharing the RTL implementation of a PE.

the global buffer can sustain higher injection/ejection bandwidth, we also simulated NoCs with multiple buses/trees and found that even with 64 buses, the design is 2× slower than the ideal.

- As the PE delay increases, normally the overall delay should increase as well, as we observe with the ideal. However, with the mesh or single bus/tree, the overall delay is almost constant demonstrating that the NoC is choked and is the bottleneck.

Area. Figure 4.2 (b) plots the area of traditional NoCs relative to the area of 64-PE array with Eyeriss [16] PEs. All numbers are from RTL synthesis with in 15nm Nangate PDK [102]. From the results, we observe that traditional scalable networks like mesh routers, prevalent across multicores, consume significantly higher area than even the compute PEs. This is primarily because routers in meshes are larger than a PE for supporting random traffic in multicore processors, not optimized for specific traffic patterns. Thus, utilizing a multicore mesh NoC inside an accelerator is not an efficient design choice as it would reduce the area available for the actual compute units, reducing overall throughput. Crossbars are known to scale horribly with number of nodes (in $O(n^2)$) and this is also apparent from our area results. Buses and the custom tree are better in terms of area.

Power. Figure 4.2 (c) shows a similar trend with power as that with area. We observe that the power consumption of meshes and crossbars is larger than that of the entire compute array, which is aggravated when we scale up the PE array, as we discuss in Section 4.5.

Our conclusion from this motivational study is two-fold:

(1) Meshes are not scalable solutions as NoCs inside accelerators. From a performance perspective, they get throughput limited when handling distributions and collections. From an area and power perspective, routers consume much higher area and power than PEs.

(2) Buses and trees are effective for an area and power point of view, but they are non-configurable, which limit the performance of accelerators. That is, they are not flexible enough to support diverse demands of accelerators to support myriad CNN topologies, mappings, and input sizes.

Link Utilization

Traditional NoCs are overly general when used inside CNN accelerators because traditional NoCs are optimized for all-to-all random traffic while traffic patterns in DNN accelerators are not all-to-all. Even in a reconfigurable accelerator that switches among several mappings, NoCs that are provisioned for uniformity can result in simultaneous under-utilization of certain links, and over-saturation of others. We perform another motivational study to illustrate this with an example.

We implement and run the row-stationary style mapping [16] on an accelerator with 8×8 Mesh NoC through all the convolutional layers of VGGNet. Figure 4.3 plots the average link utilization. We can observe that most of the mesh links are underutilized. Additionally, there is a large delta of 73% between the highest and lowest utilized link. Both of these observations indicate that there is an opportunity for specialized NoC topology with more properly distributed bandwidth.

4.2.2 Motivation Toward Specialized NoCs

To deal with the challenges for traditional NoCs and support flexible mappings in reconfigurable accelrators, we need to architect a new light-weight interconnection fabric that provides sufficient bandwidth for the various mappings with low area and power compared to the PE array. Therefore, we propose to design NoCs for DNN accelerators based on small building blocks, *micro*switch. We discuss the architecture of microswitches and present two microswitch NoC designs, Microswitch NoC-A and Microswitch NoC-B, optimized for traffic flows in reconfigurable DNN accelerators.

4.3 Microswitch NoC-A

Accelerators achieve high throughput and energy efficiency in *computation* by distributing computations to tiny processing elements, exploiting massive parallelism. Similarly, we

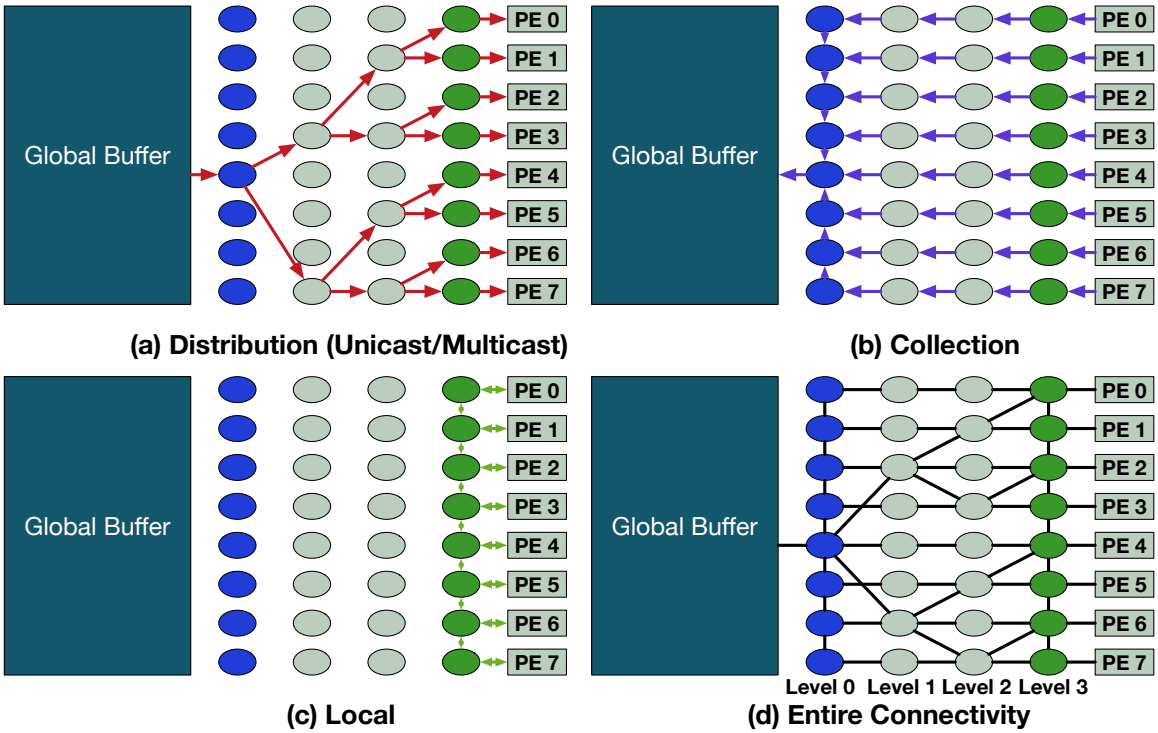


Figure 4.4: The connectivity of microswitch network for (a) distribution (unicast/multicast), (b) collection and (c) local traffic. We highlight top, middle, and bottom switches with blue, gray and green colors, respectively.

achieve high network throughput and energy efficiency in *communication* by distributing communication to tiny microswitches we propose. A microswitch consists of a small combinational circuit and up to two FIFOs; in contrast to the building blocks of traditional NoCs such as mesh routers that house buffers, a crossbar, arbiters and control. We describe the microswitch architecture in Section 4.3.2.

We design a NoC generator that aggregates multiple microswitches and connects them in our proposed topology to build a light-weight interconnect, that can be plugged into DNN accelerators. Multiple microswitches can be traversed within a single-cycle, (24 switches within a GHz at 15nm, as we show later), enabling single-cycle communication inside the NoC. We term the number of maximum microswitches can be traversed within a single cycle as MPC_{max} (maximum microswitches per cycle).

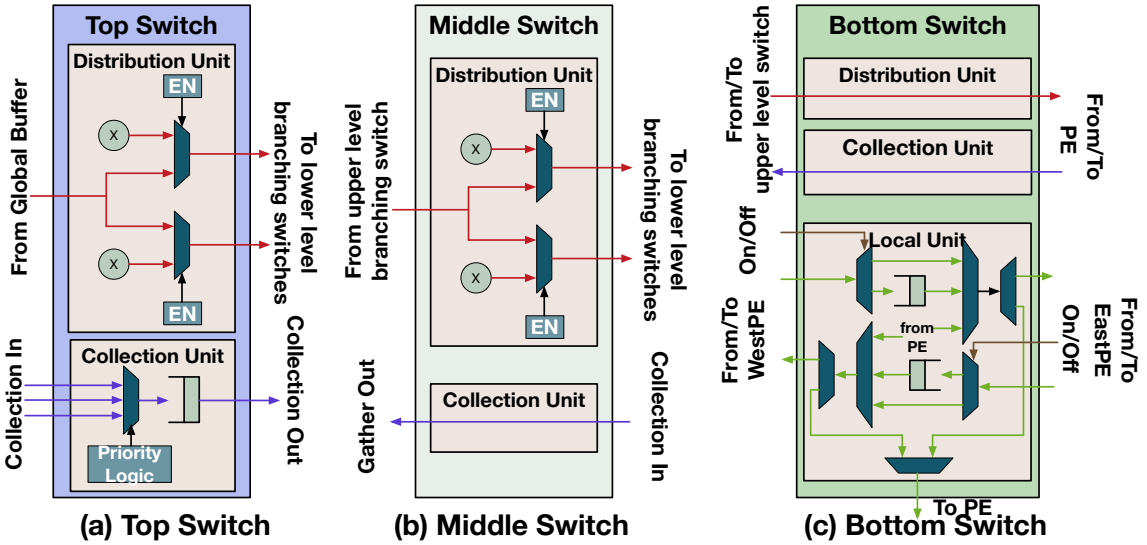


Figure 4.5: The microarchitecture of three microswitches.

4.3.1 Topology

For a N PE design, we demply $Nlog(N)$ microswitches, as shown in Figure 4.4. We divide the array into $log(N)$ levels, with N microswitches each. We numerically label the switches from the global buffer side from Level0 to Level $log(N)$. The microswitches in Level 0 are *top switches*, those in Levels 1 to $log(N)-1$ are *middle switches*, and those in Level $log(N)$ are called *bottom switches*. We layout our proposed topology over the array to efficiently handle three traffic flows that appears in any DNN accelerators described in Section 6.1: distribution (unicast and multicast), collection, and local forwarding, as shown in Figure 4.4.

Distribution (unicast and multicast)

For distribution traffic, we construct a tree structure in a microswitch array, with the root at one of the top switches, and the leaves at the bottom switches, as shown in Figure 4.4(a). This simulates the functionality of bus: delivering data to multiple destinations simultaneously within a cycle. Unlike a bus that broadcasts data to every PE, however, our design delivers data only to designated recipient PEs (i.e., a unicast or a multicast). Such selective data delivery enhances energy efficiency by suppressing redundant broadcasting; implementation is lightweight, comprising of two one-bit registers in each branching switch and control

signal propagation wires whose width is $2 \times (N-1)$ when the number PEs is N. (i.e., N-1 one-bit registers and an 2(N-1)-bit wire). We discuss the control signal generation logic in detail in Section 4.3.5. Higher throughput from the global buffer is available by simply connecting to multiple top-switches, as we discuss later in this section.

Collection

Each PE has dedicated connections up to the top switches in Level 0 via bypass links within the middle and bottom switches as shown in Figure 4.4(b), which provides high-bandwidth for collection traffic. Top switches send collection data towards one (or more) top switch connected to the global buffer's I/O port. The top switch connected to the global buffer I/O port selects one of the incoming collection flits using a round-robin-based priority logic and sends the flit to the global buffer in a pipelined manner.

Local Forwarding

For local forwarding traffic (PE-to-PE traffic), we construct a bi-directional linear network using the bottom switches, as shown in Figure 4.4(c). This network allows single-cycle traversals between any two PEs by controlling the microswitches appropriately. For example, if PE1 is communicating with PE2, PE3 with PE6, and PE7 with PE4, all of these can be supported simultaneously. We discuss this further in Section 4.3.4. The design thus minimizes the latency and maximizes the throughput of local traffic flows. The local traffic flow network is supported by the bottom switches using multiplexers and a FIFO, as discussed in Section 4.3.2. We manage the flow control using on/off reverse signaling in which each bottom switch latches incoming local flit if the buffer in the next bottom switch is not available.

Supporting Higher Bandwidth

The bandwidth of the distribution and collection networks is limited by the number of I/O ports (i.e., global buffer bandwidth) at the global buffer, as we can observe from Figure 4.4. The goal of a network architecture for accelerators has to be to make sure that the data delivery bandwidth does not become a bottleneck, which leads to PEs stalls and reduces the PE utilization. As discussed in Section 4.1, the required bandwidth depends on not only the traffic, but also the delay and context state in each PE, which the microswitch NoC generator receives as inputs. We support higher bandwidth communication from the global buffer using wider channels and/or multiple parallel networks.

4.3.2 Microarchitecture

We define the *level* of a microswitch as the number of layers between that microswitch and the global buffer, as described in Figure 4.4. Because the traffic pattern for the top (the first layer from the global buffer), middle, and bottom level of microswitch array varies, we present three types of microswitches for each.

Top switch. Top switches manage the collection and distribution (unicast and multicast), from PE to global buffer and vice versa respectively. Therefore, top switches contain two components: distribution and collection units, as shown in Figure 4.5 (a). The *Distribution unit* passes incoming flits to the branching nodes in the next level depending on the value of two one-bit control registers (one per distribution output port), determined by destinations of traversing flits. The traversal is completely bufferless, with flits branching to any one or both directions depending on the setup - unicast or multicast. The setup of the control registers is described in Section 4.3.5. The *collection unit* delivers incoming flits towards the global buffer I/O ports. There can be up to three collection flits entering a top microswitch, depending on its location, as Figure 4.4 (b) shows. A round-robin arbiter is used inside this unit. There is a an output FIFO after the arbiter to buffer the collection while it waits to win the arbitration at the next microswitch.

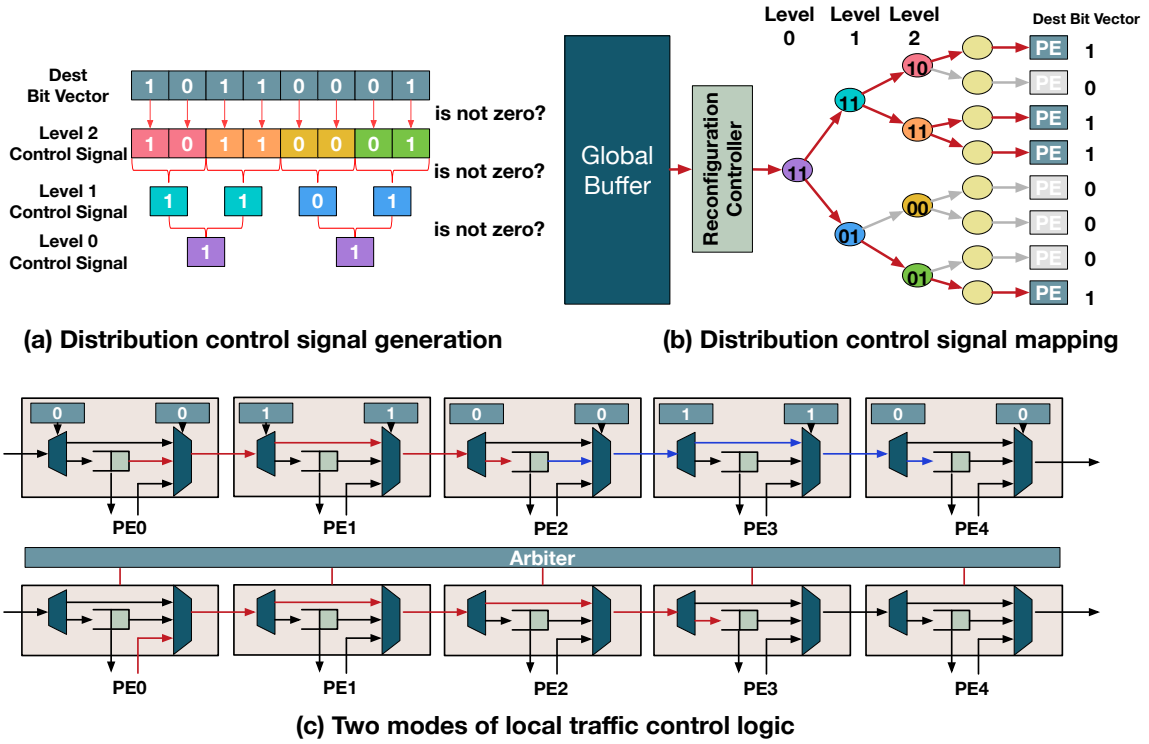


Figure 4.6: An example of distribution tree reconfiguration. (a) Control signal generation. The controller recursively tests a set of 2^k consecutive bits in a destination bit vector if is not zero until it reaches level 0. If a test bit vector is not zero, the corresponding switch is active. Therefore, the parent node switch at the lower level is active as well to provide data to the child switch. Our control logic is based on such an observation. (b) Control signal mapping for a multicast distribution. For simplicity, we only show microswitches that belong to the distribution tree. The 2-bits in each microswitch is the control register value, one for each branch of the sub-tree. For example, if the control register values are 10, incoming distribution flit is forwarded to the upper subtree in the figure. (c) Local traffic control. Mode 1 (Static) - Control register manage flow control; if the value is zero, an incoming flit stops at the bottom switch, else it bypasses. For example., this mode allows PE0 to PE2, and PE2 to PE4 communication simultaneously. Mode 2 (Dynamic) - An arbiter selects one flit and grants the flit access through multiple switches exclusively.

Middle switch. Middle switches, which belong to the level between the first and last level, manage distribution and collection traffic, as shown in Figure 4.5 (b). The *Distribution unit* is the same as that in top switches; the *Collection unit* is just a wire that simply forwards incoming collection flits toward top switches. Such a collection unit minimizes the latency of collection flits. However, if the number of PEs increases, collection flits need to traverse more number of microswitches in the middle layer. Then, we need to insert pipeline latches to meet the operating clock frequency, which is managed by our generator. Our synthesis results using NanGate 15nm standard cell library [102] shows that flits can pass 24

microswitches within a cycle (MPC_{max}) when the operating clock frequency is 1GHz, which allows 24 middle layers that cover 2^{24} PEs, a number large enough to cover state-of-the-art neural network accelerators. Note that most neural network accelerators today operate with a clock frequency lower than 1GHz and employ less than 256 PEs [60, 62, 103, 63]. *Thus our NoC can provide single-cycle traversals up to the top switches for collections.*

Bottom switch. Bottom switches, which belong to the last level adjacent to PEs, manage distribution, collection, and local traffic. The distribution and collection units are wires. The local unit consists of a small number of components: three muxes, four demuxes, two FIFOs, and combinational logic that generates the mux/demux control signals, as shown in Figure 4.5 (c). Although the number of components in a bottom switch is larger than that of components in top or middle switches, the overall overhead is not significant because the number of bottom switches increases linearly with the number of PEs. Local traffic units enable single-cycle multi-switch traversal, all the way from the source to the destination. Single-cycle multi-hop designs require extra control logic that introduces extra area and power overheads [104] to manage conflicts dynamically. We minimize such overheads by presetting microswitches to create multiple paths between PEs, as long as there are no conflicting links. We also allow flits to arbitrate for part of/the entire set of local links, like a bus.

We still require buffers in bottom switches for two reasons as follows:
(1) the buffer at the destination PE may be full; as a result the flit on the local network needs to wait.
(2) the maximum number of microswitches to be traversed may be greater than MPC_{max}. Recall that our synthesis results at 15nm demonstrate a MPC_{max} of 24 at 1GHz.

We force the bypassing flits to be latched after traversing MPC_{max} microswitches. The network interface between a PE and a bottom switch inserts a one-hot encoded bit vector that represents the number of remaining traversals. This value decreases via a simple shift

in each bottom switch during traversal, with the signal getting latched when all bits are zero.

4.3.3 Routing

For distributions, the routing is predetermined by the microswitch control logic we discuss in Section 4.3.5. The control enables broadcasts, multicasts, and unicasts within a single cycle. For collections, the route of all flows is fixed - from the PE to the global buffer. For local traffic, the the network interface (NIC) of source PEs inserts a one-hot bit vector representing the number of microswitches to traverse until the destination.

4.3.4 Flow Control

The three classes of traffic use different flow-control strategies, as determined by the switches they traverse. The overall goal is to provide single-cycle communication for all three traffic types, at the maximum possible throughput.

Distribution. For distribution traffic, we employ a customized cycle-by-cycle circuit switching technique that sets up unicast/multicast/broadcast paths that are valid for one cycle for each flit. The cycle-by-cycle circuit switching is managed by a network controller, which we describe later in Figure 4.6. The global buffer maintains credits for the input buffers in the PEs, and performs a distribution only if all destination PEs have at least one free buffer.

Collection. For collection traffic, since the traffic passes through unidirectional wires in the middle switches, no flow control is required here. Top switches, however, need a flow control for collections, since an arbitration grant plus an empty FIFO slot in the next top switch is required before a flit can be dequeued. We implement the flow control using on/off back signaling.

Local Forwarding. For local forwarding traffic, we support two schemes as follows:

(1) Static: the bottom switches are preset to enable multiple parallel circuit-switched connections between different PEs. This scheme depends on the mapping scheme across PEs and uses On/off back signaling between bottom switches. We discuss this scheme in

Section Section 4.3.5.

(2) Dynamic: part of or the entire set of local links can be arbitrated for and used like a bus.

4.3.5 Network Reconfiguration and Control

A key property of our microswitch network is cycle-by-cycle reconfigurability to support reconfigurable DNN accelerators. The reconfiguration is controlled by one-bit control registers for muxes at each microswitch, to enable single-cycle traversals across the fabric over multiple microswitches. The top and middle switches can be configured for single-cycle distributions (unicast, multicasts, and broadcasts), and the bottom switches for single-cycle local traffic. Collection network uses conventional flow-control and delivers flits in a pipelined manner, which does not require extra control signals[3] . We discuss how the control signals are generated for distribution and local forwarding networks.

Control Signal Generation.

Distribution. The reconfiguration for distribution network is controlled by two one-bit control registers in each middle and top switch that are branching nodes in the tree we construct, as the example in Figure 4.6 shows. The value of control registers indicates if an incoming data may flow toward their corresponding sub branches of a branching node in the tree. The network controller converts destination bits of a flit into control register values and sends the register values one cycle before the data flit traverses the distribution tree. The reconfiguration and the data flit traversal are pipelined so the controller inserts a data flit at every cycle. That is, while a data flit traverses the tree network, the controller generates and sends a control signal for the next data flit. Therefore, the control logic does not degrade the overall throughput.

The controller receives a destination bit vector from the global buffer, which consists of N bits (N: number of PEs) that represents valid destinations, and generates a control signal

[3]Note that collection traffic in a microswitch network have unique routing based on the source router, which does not require any control for changing routing.

that contains the value of control registers in branch switches of the distribution/broadcast tree. The control signal generation logic is based on the observation that each branching switch needs to send a flit toward a lower branch if the branch contains at least one of the valid destinations. That is, we can determine the control signal by examining two, four, and 2^k consecutive bits in a destination bit vector for the level $log(N)-k$, where N is the number of PEs and k is an integer between 0 and $log(N)$. We provide an example in Figure 4.6(a). The logic checks if an individual bit in the destination bit vector is nonzero; the results are the control signals for the last level. In the next step, the logic checks if consecutive two-bit values are nonzero; the results are the control signals for the next level. The logic repeats to double the size of test consecutive bits and check if each chunk is nonzero until the test bit size covers the half of the destination bit vector. If the number of PEs is not a power of two (i.e., number of PEs $< 2^k$), the logic regards the destination bit width as 2^k and pads zeros for invalid destinations.

Local Forwarding Network. On the local network, we provide the ability to partition the set of local links into single-cycle circuit-switched paths between any two PEs (subject to the number of switches being less than MPC_{max}). The local network configuration is done across larger time epochs rather than every cycle. For instance, for CNNs, this is done at the start of every convolutional layer. Since the network controller manages delivery of distributions, it also knows which PEs will communicate with which other PEs, and accordingly tries to provide neighbor-to-neighbor communication as much as possible, which can be supported in parallel, as shown in Figure 4.6(c). Each bottom microswitch has 2-bits to determine whether incoming flits need to be forwarded to the next microswitch or stop. Flits that stop at a bottom switch are read by the appropriate PE if the destination matches. Thus the bottom switch allow the local links to form configurable buses of different lengths.

If partitioning the bus statically is not possible for handling all local communication flows simultaneously, say for fully-connected layers of CNNs, some/all bottom switches

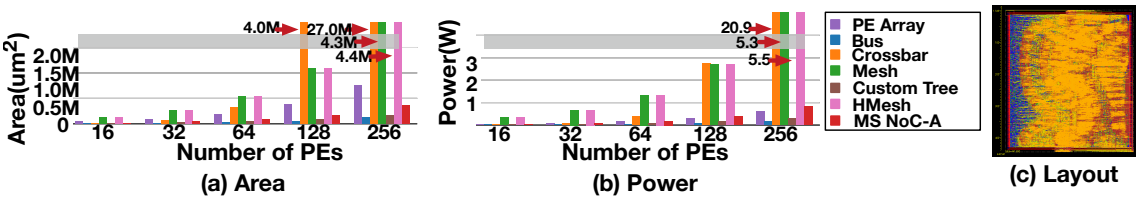


Figure 4.7: Post-synthesis area/power estimation and layout on ASIC. The ASIC chip dimension is 440x440um.

operate in a forward mode and the local links behaves like a bus via dynamic arbitration, as shown in Figure 4.6(c). We also enable part of the local links to operate like an arbitrated bus, and the remaining to be statically configured. This is all managed by the reconfiguration controller.

Control Signal Mapping

We utilize a separate control plane to configure each microswitch. Recall that each switch has a 2-bit configuration state. The number of bits in the control plane is a trade-off with reconfiguration time, and multiple implementations can exist. We list two implementations, which are in trade-off space of reconfigurable time and hardware cost:

Dedicated. We use $2 \times NlogN$ wires, to enable cycle by cycle reconfiguration. As an energy optimization, the controller only sends bits to switches that need to update their configuration. A challenge with this design is that the configuration plane may become too wide at large PE counts.

Ring. We also support an alternate design for the control plane where all switches are linked via a configuration ring (analogous to scan chains today) to carry a switch id and the 2-bit configuration. The controller sends configurations for each switch multiple cycles in advance, keeping the delay of traversing the ring in mind. This is possible since the dataflow is fixed after the mapping is complete.

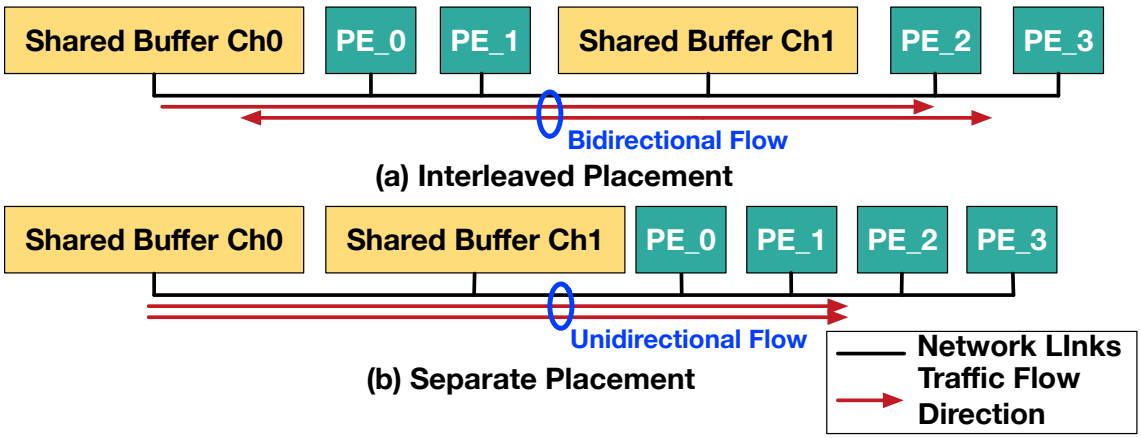


Figure 4.8: PE and shared buffer channel placement with distribution traffic. Interleaved placement involves bidirectional flow, which requires more complicated hardware than unidirectional flow. Collection traffic has the same trends with traffic directions reversed

4.4 Microswitch NoC-B

We propose an alternative set of microswitches with more focus on composability while Microswitch NoC-A is more specialized for providing high bandwith for DNN traffic.

4.4.1 Topology

We explore the topology of Microswitch NoC-B beginning with the placement of the PE and shared buffer channel ports. The placement of PEs and shared buffers can be either interleaved or separate, as illustrated in Figure 4.8. For distribution and collection traffic, interleaved placement requires bidirectional links while separate placement requires only unidirectional links. To reduce the hardware complexity from bidirectional link support, we choose separate placement in our topology.

For the dimension of the topology, we choose a 2D topology because a 1D topology involves hot-spots that require tremendous bandwidth as shown in Figure 4.9 (a). Within the 2D topology, we can implement either a direct or indirect topology. Direct topology, in which data from shared buffer channels traverse directly to PEs contains routers or switches, requires bidirectional links as shown in Figure 4.9 (b). However, indirect topology does not involve bidirectional link so it contains potential optimization opportunities to reduce hardware overhead. Therefore, we select the 2D indirect topology for Microswitch NoC-B.

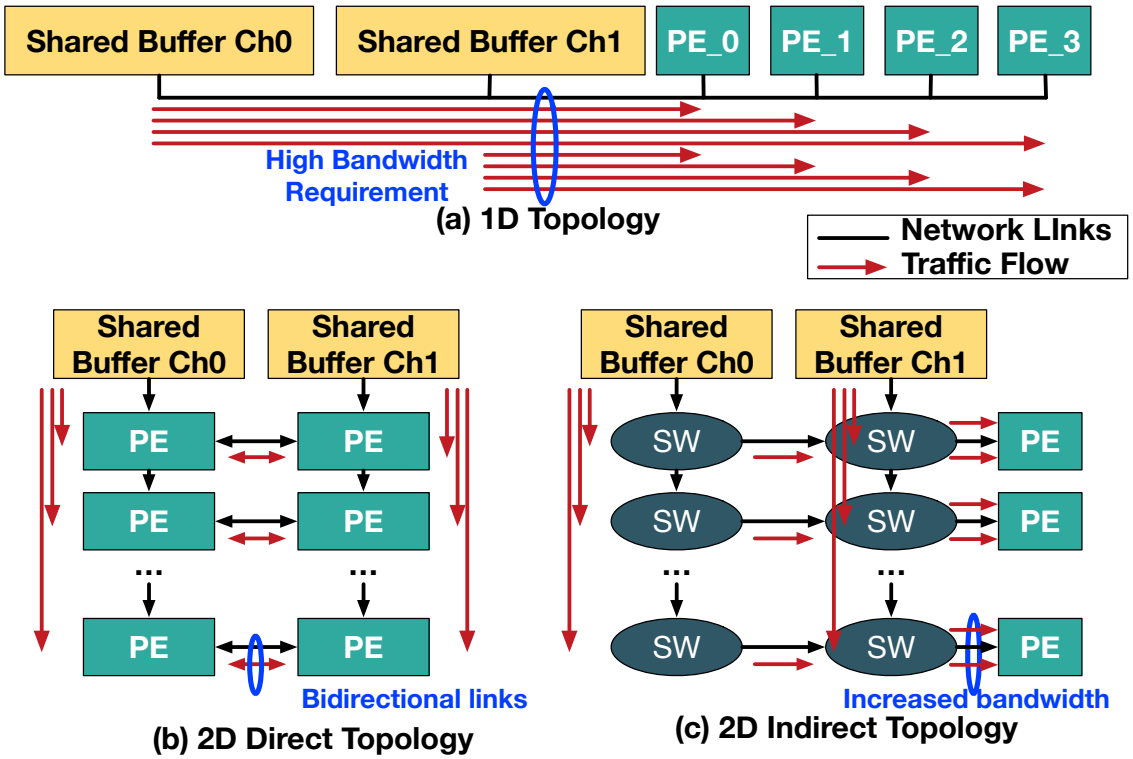


Figure 4.9: Dimensions of topologies with distribution traffic. 1D topology involves hot-spots that requires high bandwidth in a certain link while 2D topology distribute traffic in a better way. 2D indirect topology includes only unidirectional links but require more bandwidth in links near PEs. However, it is much less than bandwidh requirement from 1D topology.

To design a new NoC topology for reconfigurable accelerators, we consider the three traffic classes discussed in Chapter 4 and design separate networks for each class to specialize designs for each traffic. It is important to note that any datatype (input activation/filter weight/partial sum) can use any of the three networks, depending on the dataflow being mapped. We discuss this later in Section 4.4.3. This is in contrast with prior works [16, 71] that use separate networks per data type rather than per data pattern.

Distribution Network:. Because distribution is inherently one-to-many communication, multicasting support is a crucial feature of NoC for distribution traffic. In addition, in some accelerators [16], the PE array requires many multicasting at the same cycle as a result of data reuse optimization. Therefore, providing enough bandwidth for many-mutlicasting is necessary to support such dataflow. We can realize such a network by using a structure described in Figure 4.10. In the topology, each shared buffer write ports can inject write data to all the PEs. If a shared buffer port sends multicasting data, each switch in the same

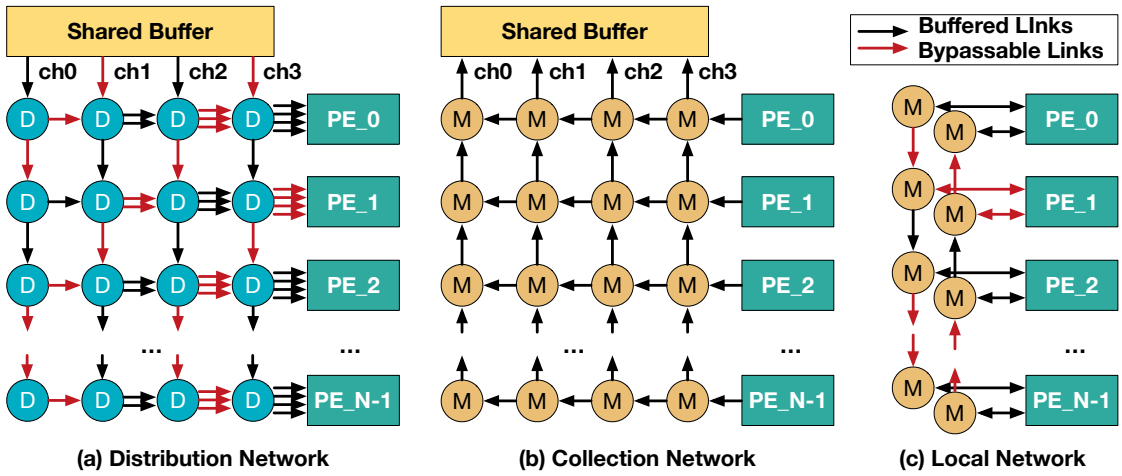


Figure 4.10: Topology of proposed network-on-chip architecture. Ch indicates channels of the network.

column of the shared buffer port either (1) duplicates data from its upper port to the right and bottom port, (2) just forwards data to its bottom port, or (3) blocks the data and removes it from the switch array based on the destination vector.

Collection Network: . For collection, the shared buffer write bandwidth determines the collection network bandwidth. In other words, even if a NoC provides higher bandwidth than shared buffer write bandwidth, the effective bandwidth is still rate-limited. Therefore, we avoid extra overhead from over-designed NoC by matching the bisection bandwidth of the collection network with the shared buffer write bandwidth. Figure 4.10 (b) describes such a network whose horizontal bisection bandwidth of the proposed network exactly matches the shared buffer write bandwidth.

Local Network: . Local traffic is many multiple data forwarding between two PEs in a certain offset. When the offset is not one, typical approaches that implement direct links between all the adjacent PEs require more than one cycle because data needs to stop at each PE. To address such a challenge, we adopt single-cycle multi-hop network design, SMART [105] for the local links, similar to the local links in [29]. Unlike the previous work involves complicated routers for SMART links, we implement the SMART network using an array of merge switches. We design a bidirectional linear SMART network that supports multiple data forwarding between non-adjacent PEs. We parameterize the bandwidth of

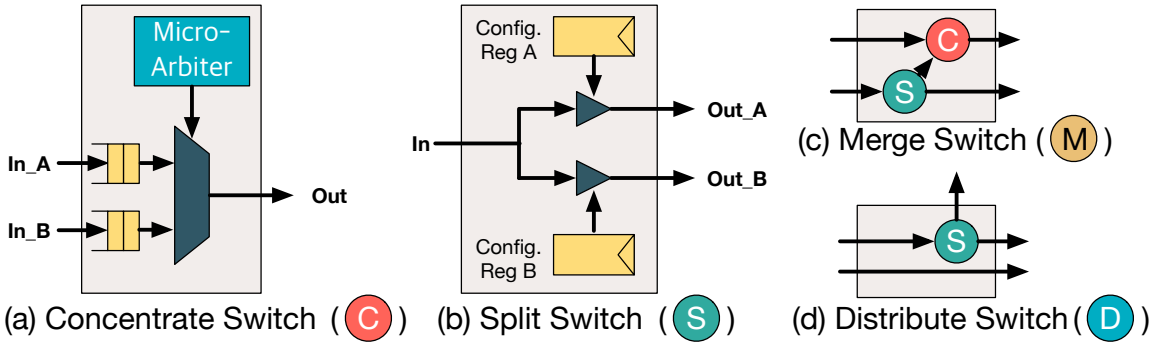


Figure 4.11: The microarchitecture of Microswitch-B. Distribute and Collection networks use Distribute (D) and Merge (M) microswitches shown in (c) and (d). These are built using primitive 2:1 and 1:2 switches (Concentrate and Split) shown in (a) and (b))

the linear SMART network to resolve link conflicts when non-adjacent PEs perform local communication.

4.4.2 Microarchitecture

Fine-grained Switching Idioms

Many NoC substructures such as crossbar or matrix arbiter have non-linear overhead over the degree of connections per router or switch. For example, the area and power overhead of an N-input-N-output crossbar switch is $O(N^2)$ because it requires N^2 connection points and wire segments. A common approach to address such super-linear overhead is decompose a large structure into multiple small structures [**dragonfly˙router**]. We apply this approach to design the switches used in the Microswitch NoC-B topology, because the degree of connectivity is expected to be in the hundreds. We define two key switching idioms that are required to support the Microswitch NoC-B topology.

- **2:1 (concentrate) switch** (Figure 4.11(a)) contains a MUX and a 2:1 matrix arbiter that implements a fair arbitration policy based on a last used register. Because only one of two input data can win the output port, the concentrate switch requires back pressure at each input port. We implement this using simple 2-deep FIFOs and rely on inter-switch flow control to reduce the area and power overhead as much as possible.

- **1:2 (split) switch** (Figure 4.11(b)) contains two control registers that configure the flow of input data. Split switches can block incoming data, send incoming data to one of the output ports, or send incoming data to both of the output ports. Split switches manage multicasting of incoming data to two directions.

Microswitch-B

Using the primitive switches described above, we design two microswitches (distribute (*D*) and merge (*M*)) to provide bandwidth distribution across the Microswitch NoC-B indirect topology (Figure 4.10) at extremely low area, power, and latency overheads, addressing the traditional NoC challenges discussed in Section 4.2.

- **Merge (*M*) switch** is built using one split and one concentrate switch, as shown in Figure 4.11(c). A *M* switch in row *R* (in the same row as PE_R) in the distribution network, manages a link toward shared buffer shared between collection data from PEs with ID larger than *R* and PE_R. If PE_R sends collection data via a channel other than the merge switch manages, the merge switch forwards the data to the next merge switch in the same row that manages different channels. In the local network, *M* switch manage linear network links shared between local data from remote PEs and injection data from the local PE.
- **Distribute (D) switch** is built using a distribute switch and a direct bypass wire, as shown in Figure 4.11(d). A *D* switch in column *C* manages multicast from the shared buffer channel *C* and forwards (potentially multicasted) data from other channels with ID smaller than *C*. For example, the upper-most and right-most *D* switch in Figure 4.10 manages multicasting from channel 3, and forwarded multicast data from channel 0..2 to PE_0.

4.4.3 Routing Algorithm

In the Microswitch NoC-B, the routing algorithm is minimal and deterministic for all the subnetworks presented in Figure 4.10 based on destination PEs of each data and channel ID. Any data type can be inserted into any of the subnetworks and will be delivered to the right physical queue in the PE or L2 buffer .

When the distribution network(Figure 4.10 (a)) receives data from one of the shared buffer's channel, the data traverses straight down (meaning orthogonal direction to the PEs) toward the lower switch rows until it reaches the row with the last destination PE. On the way to the last row, Distribute switches duplicate the data toward other destination PEs. Therefore, the routing algorithm for the distribution network is YX routing with duplicating data at each row with a destination PE.

In the collection network (Figure 4.10 (b)), each data traverses horizontally until it reaches the column with the same channel ID as its channel ID. When the data arrives at the merge switch in the target column, it makes a turn to the vertical links to traverse toward shared buffer. Therefore, the routing algorithm for the collection network is XY routing.

In the local network (Figure 4.10 (c)), a PE injects data toward one of the linear merge switch networks based on the direction of the destination PE. Injected data traverses directly to the destination PE via a linear network without any misrouting. Therefore, the routing algorithm for the local network is a minimal linear routing algorithm.

4.4.4 Flow Control

We apply credit-based flow control for all the networks. We integrate the NoC flow control to correspond to PE data requests. The system starts with zero credits for all VCs at the shared buffer, which which implies no buffer availability. When a PE sends credit signal to the shared buffer, credit signal propagation constructs a temporary route from the shared buffer to the PE that allows data traversal of the same number of sent credits.

As we discussed in this section, we believe Microswitch NoC-B is a light-weight

Table 4.1: A summary of evaluation configuration

Network	Bus, Custom tree, Crossbar, Mesh, Hierarchical Mesh (4 clusters, 1X or 2X BW at GB), Microswitch NoC-A, Microswitch NoC-B
Language	Bluespec System Verilog (BSV) [106]
Technology	15nm NanGate PDK [102] (synthesis) and 28nm technology (PnR)
Mappings	weight-stationary (without local accumulation) and row-stationary styles
Application	Alexnet (CNN) [1] and VGGnet [2]

interconnect that distribute bandwidth for CNN dataflows while maintaining flexibility. To verify our belief, we implement the accelerator model in RTL and evaluate our model in performance, area, power, and energy for multiple dataflows.

4.5 Evaluations

In Table 4.1, we present the evaluation methodology and configurations. In addition to traditional NoCs, we also implement and evaluate the performance of hierarchical designs, which are popular in recent DNN accelerators [62, 101, 71]. We use a hierarchical mesh with four clusters (e.g., for 64 PEs, each cluster contains 16 PEs).

4.5.1 Area and Power Scalability

Figure 4.7 presents the post-synthesis area and power results of microswitch NoC-A compared to a traditional NoCs. The first stark observation is that the mesh adds too much overhead, both in terms of area, and power, compared to even the PE array. The crossbar area and power are reasonable at 32-64 PEs, but it shoots up at large PE counts. The mesh and crossbar consume 7.4X more power and 7.2X more area compared to the PE array at 256 PEs. The bus[4], tree and microswitch array are the most scalable for area and power. On average, the microswitch array consumes 47.8% lower area and 39.2% lower power than all baselines. Assuming a 512B SRAM in each PE [63], we find that a microswitched based accelerator can house 2.32X more PEs than a mesh in the same area.

[4]Note that this is a post-synthesis result that does not take into account the RC of the final bus layout. Thus the observed power consumption is somewhat under-evaluated.

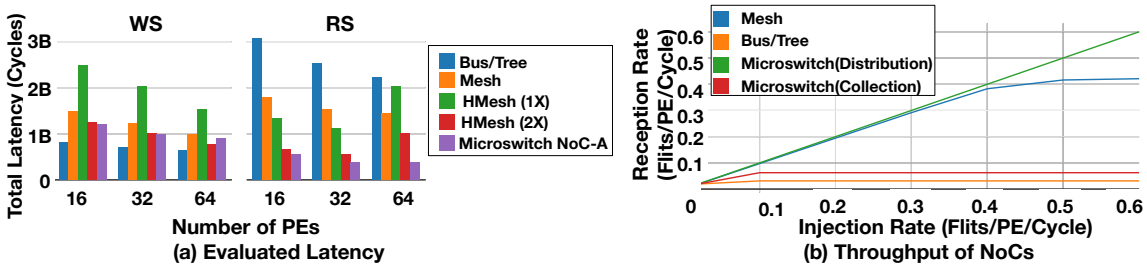


Figure 4.12: (a) Total latency of each accelerator and NoC combination for entire Alexnet. (b) Throughput evaluation of mesh, bus, and microswitch network with 32 PEs and randomized synthetic traffic.

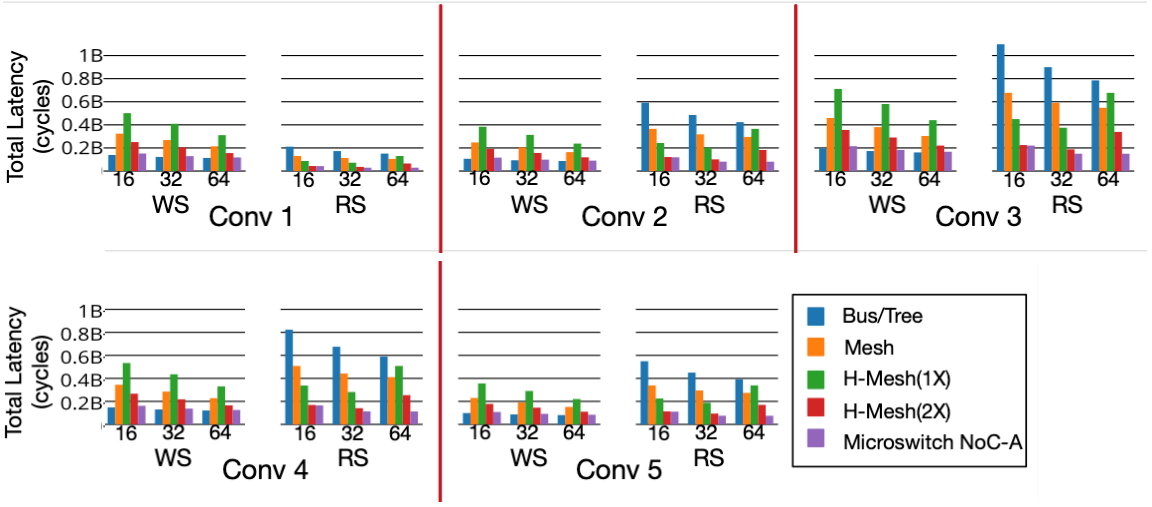


Figure 4.13: Runtime of WS/RS accelerators for each Alexnet conv layer. The number below each group of bars represents the number of PEs.

4.5.2 Throughput and Latency Scalability

Figure 4.12 (a) presents the total latency for running Alexnet in WS and RS accelerators with 16, 32, and 64 PEs. Since multicast-distribution is dominant in weight-stationary traffic, as Figure 4.1 shows, the bus and tree performs well with WS accelerators. However, as RS accelerators involve local traffic, the micro-switch network performs the best because it exploits the local traffic network between the bottom switches. Mesh performs the worst in every case because it needs to serialize all the distribution traffic. An optimization that clones a distribution flit in each router is feasible, but such an optimization demands more area and power. Considering the area and power overhead of mesh is already prohibitively high, as we discss in Section 4.5.1, it is not practical even if such an optimization were to be applied. The HMesh with 2X bandwidth at the global buffer performs better than the mesh,

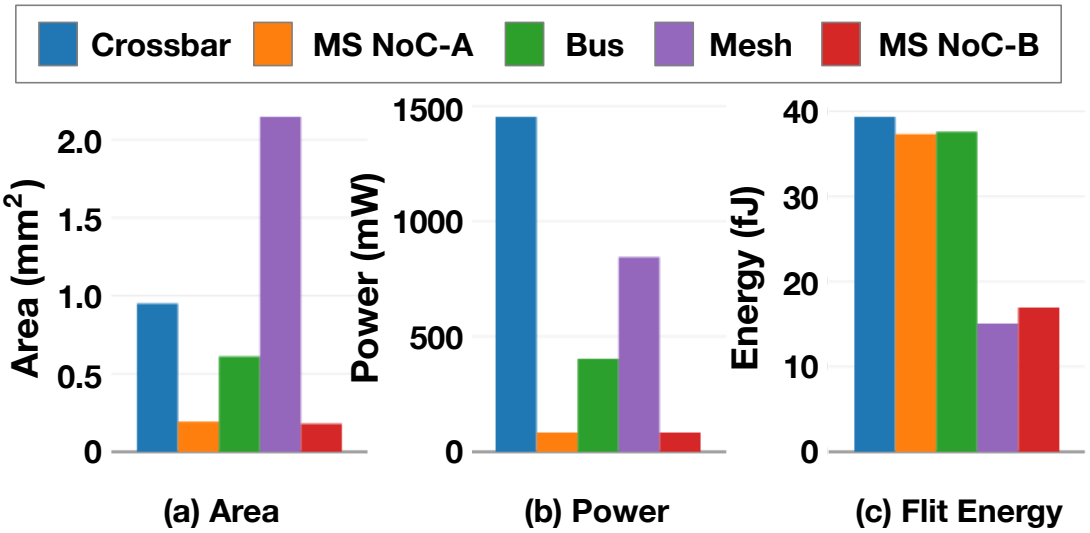


Figure 4.14: Post-Place&Route (a) area and (b) power consumption of bus, Miscroswitch NoC-A (MS NoC-A), mesh, and Microswitch NoC-B (MS NoC-B) with 64PEs. (c) shows the energy for a flit traversal in the NoCs we compare.

but is still worse than the bus and microswitch which have 1X bandwidth due to the lack of multicast support. 'For RS traffic, the HMesh had worse performance because of mapping inefficiencies caused by the fixed size of clusters.

In Figure 4.12(b), we compare the performance of the networks with synthetic random distribution/gather traffic. The performance of the microswitch distribution network scales linearly without saturating as it guarantees single-cycle traversal to multiple destinations via the single-cycle multiple-hop network. The microswitch gather network saturates early due to heavy congestion at the link going into the GB, and we recommend using multiple gather networks or wider links at the top switches to enhance throughput. The bus and tree networks saturate very early.

Figure 4.13 shows the performance breakdown of the NoCs for running each layer of AlexNet. The micro-switch fabric provides the lowest runtime, a 49% savings on average across all NoCs, as it eliminates the distribution and/or gather bandwidth bottlenecks present in other NoCs.

4.5.3 Area, Power, and Energy of Microswitch-A and -B

We compare the area, power, and flit traversal energy of crossbar, microswitch-A, bus, mesh, and microswitch-B. We observe that microswitch NoC-B required the least area and power among five NoCs we compare. On average, microswitch NoC-B consumes 81.54% lower area and 88.0% lower power compared to other NoC options.

We observe that microswitch NoC-B require 7.0% less area but 0.1% more power compared to microswitch NoC. However, the flit traversal energy of microswitch NoC-B for each flit is 54.5% less than that of microswitch NoCs. This is because microswitch NoC-B does not have long wires unlike microswitch NoC has long wires for implementing a tree structure. On average, microswitch NoC-B requires 47.6% less energy for each flit traversal.

Based on the multicast functionality, the behavior of distribution network of microswitch NoC-B is similar to multiple-bus, which is used in other accelerators [16, 68, 71]. Bus is one of the most area and power-efficient NoC but bus can easily be congested by multiple requesters [29]. Also, bus always broadcast data even if the recipient is only one while microswitch NoC-B distribution network selectively activate links only if they are necessary for the desired destinations. Such differences resulted in 54.9% less energy of microswitch NoC-B compared to bus.

4.5.4 Bandwidth Distribution of microswitch NoC-B

Figure 4.15 presents the link utilization of distribution network connected to 64 PEs and four shared buffer channels with Eyeriss-style dataflow (row-stationary). Compared to the link utilization plot of mesh presented in Figure 4.3, we can observe that load to each link is distributed better. Based on the heatmap and area/power evaluation, we can conclude that microswitch NoC-B arrange links in an optimized way for CNN accelerators. By such optimization microswitch NoC-B could achieve better performance than mesh using less hardware resources.

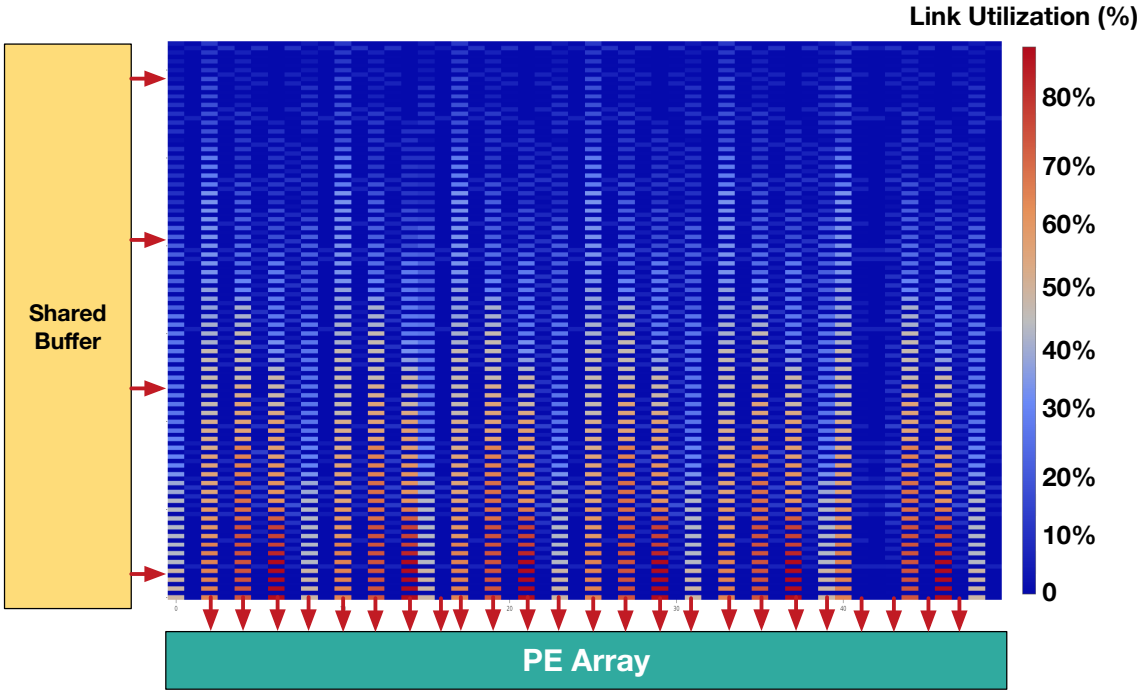
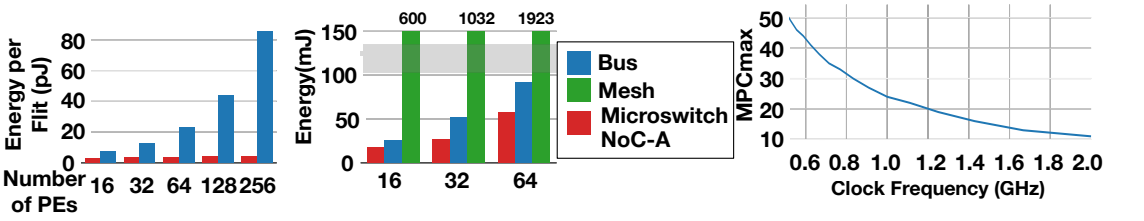


Figure 4.15: Link utilization heatmap of microswitch NoC-B's distribution network over 64PEs.



(a) Energy for each flit (b) Energy of an RS Accelerator (c) MPCmax Value and Clock Frequency

Figure 4.16: (a) Energy consumption for single flit traversal. (b) Total network energy for entire Alexnet convolution layers using an RS accelerator (c) MPC_{max} over clock frequency values.

4.5.5 Energy consumption

Since a bus always broadcasts flits to the PE array, it requires more energy for each flit. The worst case of such an inefficiency is unicast that has only one destination but bus consumes energy for broadcast. More number of PEs aggravate the energy inefficiency of bus, as Figure 4.16 (a) highlights. The amount of energy required for single flit traversal affects the overall energy consumption of entire computation. The total energy consumption for Alexnet convolution layers in Figure 4.16 (b) shows the micro-switch NoC being the most efficient in terms of overall energy as it activates only the required minimal links for each flit traversal, for both distributions and gathers.

In summary, we can observe that the micro-switch network performs well on all metrics - latency, throughput, area, power, and scalability, when used inside a neural network accelerator, while traditional NoCs fail on one or more of these fronts.

4.5.6 Bottom switch bypass for local traffic

Depending on the operating clock frequency, the number of bottom micro-switches a local traffic flit can traverse within a cycle i.e., MPC_{max}, varies, as shown in Figure 4.16 (c). The MPC_{max} value affects the throughput of local traffic network based on the source-destination pattern. If an accelerator design requires end-to-end local traffic, then the delay of such local traffic flits is the number of PEs divide by MPC_{max}. However, assuming that the neural network mapping algorithm did a good job mapping communication PEs close to each other, such a worst case would be rare, and we expect most local traversals to take a single-cycle leveraging the single-cycle over MPC_{max}-hops feature of our micro-switch array. For example, a PE in an RS accelerator requires partial sums to traverse to an adjacent PE in the same column of the PE array [63]. We can construct a column-first linear bottom switch network for it, and the number of bypass hops toward the destination is always one in such a configuration. This configuration works with either of the control logic we discussed in Section 4.3.5.

4.6 Summary

In this chapter, we presented a novel approach to designNoCs for reconfigurable DNN accelerators that consists of configurable light-weight micro-switches. The micro-switch network is a scalable solution for all the four aspects - latency, throughput, area, and energy - while traditional NoCs (bus/mesh/crossbar) only achieve scalability for some of them. We also provide a reconfiguration methodology to enable single-cycle paths over multiple micro-switches to support dynamism across neural network layers, mapping methodologies and input sizes.

Based on the high bandwidth and the low latency, Microswitch NoC minimizes PE stalls due to the delay of operand arrival at each PE, which maximizes operational utilization of PEs. However, even if an accelerator provides near 100% operational utilization, if a mapping does not utilize all the PEs, overall utilization of PEs will be constrained. Such underutilization based on mapping often occurs because of the size mismatch between PE array and computation/data tiles of a mapping. In the following chapter, we propose a hardware solution for minimizing such mapping underutilization that achieves near 100% mapping utilization ($(\frac{\text{The number of PEs with computation assigned}}{\text{(The number of PEs)}}$).

CHAPTER 5

MAERI: A RECONFIGURABLE IN-NETWORK-PROCESSING ACCELERATOR FOR IRREGULAR NEURONS IN DNNS

In the previous chapter, we discussed the microswitch NoC which provides high bandwidth and scalability for reconfigurable DNN accelerators. Microswitch NoC facilitates to achieve high operational utilization, which is the average number of active PEs (impacted by PE stalls of PEs) by minimizing PE stalls due to communication delay. However, if a mapping and the PE array dimensions do not match, the mapping utilization, which is the number of PEs with assigned operations divided by the total number of PEs, decreases. Since the mapping utilization defines the roof-line utilization of a PE array (i.e., overall utilization when operational utilization is 100%), it is also critical to maximize mapping utilization. As a solution, in this chapter, we discuss a DNN accelerator architecture that provides near 100% mapping utilization for arbitrary mapping.

5.1 Challenges for Supporting Flexible Dataflows

As we discussed in Chapter 4, traffic in DNN accelerators can be categorized into three classes (distribution, collection, and local forwarding), and efficiently supporting those traffic patterns is the key requirements for on-chip interconnect, or NoC. However, generic all-to-all NoCs like a crossbar or a mesh is extremely area and power inefficient [29, 63] for an array of 100s of tiny PEs, and as a result almost every DNN accelerator has used a hierarchy of buses [63, 21, 60] and/or trees [21, 72]. For instance, Eyeriss [63] uses buses to connect 12 PEs together in a row, and 14 rows are connected together by another bus. SCNN [20] creates clusters 4x4 PEs with adder trees internally, and an external bus connecting to on-chip SRAM. The size of each cluster is often determined by the nominal size of the filters (3×3 or 4×4), and the buses/trees optimized for data distribution and collection from

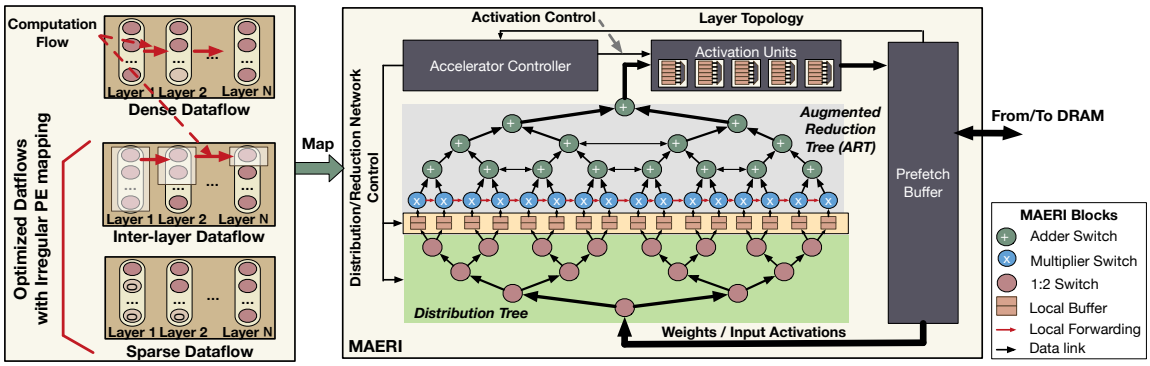
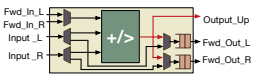
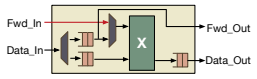
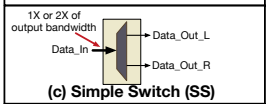
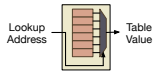


Figure 5.1: An overview of MAERI. MAERI is designed to efficiently handle CONV, LSTM, POOL and FC layers. It can also handle cross-layer and sparse mappings. We implement this flexibility using a novel configurable interconnection network topology within the accelerator.

these. This rigid structure restricts arbitrary filter sizes or cross-layer dataflows from being supported in these designs. Moreover, when optimizations such as sparsity are introduced, the filter sizes can vary dramatically while the size of the PE clusters is fixed, leading to inefficient utilization as we demonstrate in this work.

FPGAs have been popular substrates to evaluate various dataflows [72, 107, 71] as they provide the flexibility of running different dataflows based on their reconfigurable substrate. A recent work explored FPGA design optimization for DNN [72] demonstrates that the nominal tiling factor for CNN computation can vary layer by layer and generates optimized RTL for every layer. Fused CNN [107] augments this approach to support cross-layer dataflows on FPGAs. However, while FPGAs vs. ASICs for DNNs is an ongoing debate, with the flexibility being touted as a reason to prefer the former, the area, timing, and power-efficiency of ASICs is still the strong case for ASICs.

The goal of this work is to provide the flexibility afforded by FPGAs today in terms of flexible dataflow support to ASIC accelerators. Our key idea is to use a homogeneous, rather than hierarchical, design and provide flexibility within the NoC topology to create virtualized clusters of arbitrary sizes at runtime.

Building Block	Description
 (a) Adder Switch (AS)	The AS is an adder augmented with a tiny switch that enables us to map arbitrary adder trees over our non-blocking reduce network called ART. It also contains comparators for pooling operations.
 (b) Multiplier Switch (MS)	The MS is a multiplier augmented with a tiny switch that is used for local data forwarding. It is used by CNNs for generating partial sums from weights and input activation values, and by RNNs for generating gate values, input activations, and previous output activations.
 (c) Simple Switch (SS)	The SS provides 1:2 switching functionality in the distribute network's chubby tree nodes.
 (d) Configurable Look-up Table (LT)	LTs implement activation functions such as *sigmoid* or *tanh*. We load all the necessary functions for a neural network and change its target function in run time based on the configuration generated by *MAERI*.

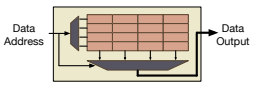
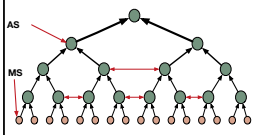
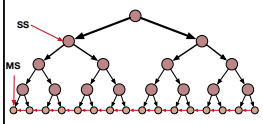
Building Block	Description
 (e) Prefetch Buffer (PB)	The PB works like a cache memory between DRAM and the computation units. We implement this as a private scratchpad. Because the characteristics of a prefetch buffer would differ based on SRAM technology library and they are usually commercial libraries, we provide a default multi-banked implementation using flip-flops.
 (f) Reduce Network (RN)	The RN is a network structure for reduce and collection operations. It is based on a new adder tree structure, augmented reduction tree (ART) we propose in this work. ART facilitates mapping multiple configurable non-blocking adder trees and minimizing inactive multiplier switches.
 (g) Distribute Network (DN)	The DN is based on a chubby-tree structure, which is a tree-based network with wider link bandwidth in higher levels of a tree. We exploit abandunt bandwidth at high level links to enable multicast functionality, which is one of the most common traffic patterns in DNN accelerators.

Figure 5.2: The microarchitecture of building blocks used in MAERI and description of them.

5.2 Building Blocks

Figure 5.1 provides an overview of the MAERI microarchitecture. We use a homogeneous design with plug-and-play building blocks that are listed in Figure 5.2. A prefetch buffer (PB) serves as a cache of DRAM, and stores input activations, weights, intermediate partial sums that could not be fully accumulated, and output activations. Lookup Tables (LTs) implementing activation functions are located between the root of the reduction-tree and the PB. The secret sauce that enables our flexibility is two-fold:

(i) We augment the multipliers and adders with configurable switches, calling them *multiplier switch (MS)* and *adder switch (AS)* respectively, enabling MAERI to optimize for the collective communication patterns.

(ii) We use two configurable interconnection networks - a distribution network, built using tiny *simple switches (SS)* sends inputs to the MSes from the PB. A reduce+collect network, built using ASes, sends outputs back to the PB via activation units implemented with look-up tables.

The entire accelerator is controlled by a programmable controller which manages reconfiguration of all three sets of switches (MS, AS, and SS) for mapping the target dataflow.

We design two novel interconnect topologies specialized for the distribution and reduc-

tion+collection flows, which we describe next. These networks can be tuned to provide full non-blocking bandwidth to the compute blocks, but can be pruned to reduce the bandwidth if required.

5.2.1 Data Distribution Network

The data distribution network of MAERI sends input activations and weights from the PB to the MSes. It ensures full non-blocking bandwidth to all the multipliers.

Topology

We use a binary-tree as our base topology for distributions as it is multicast friendly, and augment it with (a) chubby links for supporting higher-bandwidth from the PB, and (b) forwarding links, to implementing store-and-forward multicasts for CNNs. We describe these next.

Chubby Links from Root. A fat-tree supports 2x bandwidth at every level from the leaves up to the root, and is a classic topology for providing non-blocking bandwidth, and is used extensively in datacenter networks [108]. However, such a topology is infeasible to build on-chip since the bandwidth requirement at the root would require too many wires and ports at the PB, resulting in significant area and power overheads[1]. To address this design-challenge, we propose to use a **Chubby tree**, where the bandwidth at the upper levels is either 2x or the same as that at the lower levels. For example, in Figure 5.3, the bandwidth of link level 0 is twice of that of link level 1, but the bandwidth of link level 2 and 3 is 1x.

We size the bandwidth at the root to equal that of the PB, and taper the width going down the tree, providing non-blocking flows till that level. Once bandwidth becomes 1x, the bandwidth for the rest of the levels are compensated for by adding local buffers at the MSes to hide the communication delay due to multiplexing of links at the 1x levels.

[1]For a fat-tree network, 256 MSes would require a 256 ported SRAM.

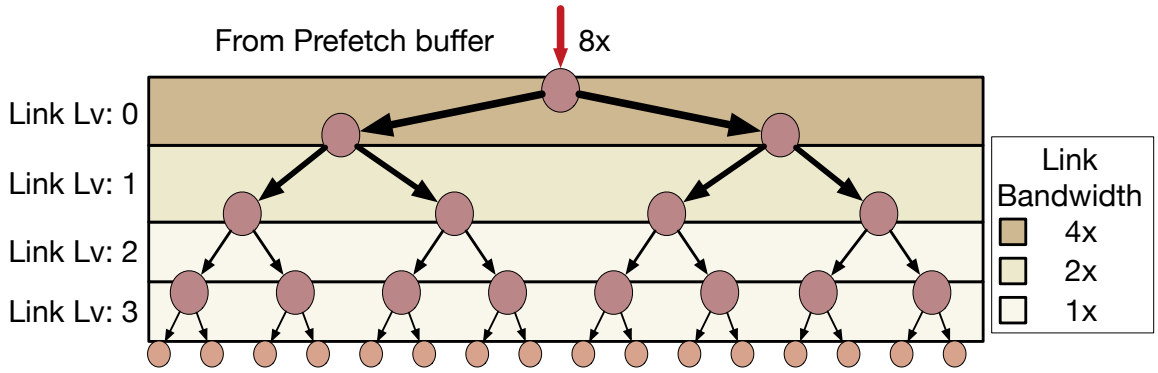


Figure 5.3: Example of chubby distribution tree. Leaves are multiplier switches. Other nodes are simple switches without compute units.

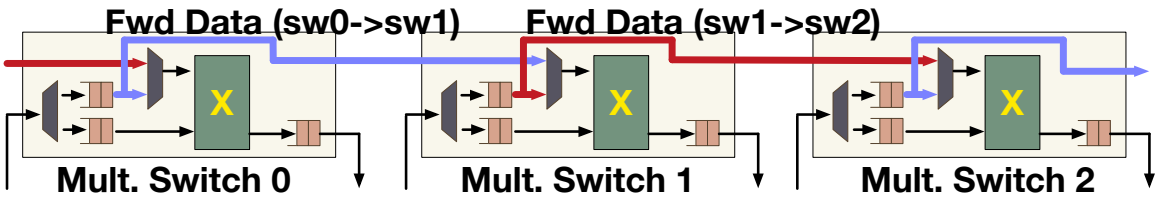


Figure 5.4: Data forwarding links (thick arrows) facilitate data reuse between adjacent multiplier switches.

Forwarding Links at Leaves. MAERI provides local data forwarding links (FL) between the leaves (i.e., adjacent MSes) in the distribution network, as shown in Figure 5.4 to provide store-and-forward multicast for input activation values. We highlight their use for CNNs in Section 5.3. The FL's are unidirectional since MAERI maps data over the multiplier switches in order, so input activation values flow in one direction.

Microarchitecture, Routing and Flow-Control

The distribution tree nodes use a *simple switch (SS)* whose microarchitecture is shown in Figure 5.2. SSes are bufferless demuxes with a select line that is set by the input data directly. SSes to chubby links are direct connections, since no bandwidth sharing is being done.

Since the topology is binary-tree based, input data is source routed, with a bit to choose between the left and right paths at each switch. Since the SSes are bufferless, the flow-control is end to end between FIFOs at the MSes and the PB. Also, *we provide single-cycle traversals from the PB to the leaves (MS)* for every piece of data.

5.2.2 Data Reduction and Collection Network: ART

Binary trees are well-suited for performing reductions and have been used in prior DNN implementations [60, 62, 65, 73, 68] to implement adder trees within the PE clusters described in Section 5.1. However, they have a key inefficiency: the fixed topology of a tree is inherently inefficient whenever the number of partial sums to be accumulated is smaller than the width of the adder tree, as illustrated in Figure 5.5 (a). Suppose there are 16 multipliers, all connected via a 16-node binary reduction tree. Each node in the reduction tree is an adder. This tree is perfect for performing a reduction for a 16-input neuron. However, suppose we map three neurons over these multipliers, each generating five partial sums, as Figure 5.5(a) shows. Each neuron requires four additions to generate an output, so the total additions required is 12. The reduction tree contains 16 adders, which should be sufficient to perform the additions for all neurons in parallel. However, the four links in red are shared by multiple neurons, limiting the tree from generating all three outputs simultaneously.

More formally, the challenge pertains to mapping arbitrary number of reduction trees over an underlying fixed topology. To provide flexibility to support any mapping, this needs to be addressed, otherwise there would be a drop in utilization. We solve this challenge by proposing a new tree topology called an Augmented Reduction Tree (ART).

Topology of ART

The ART is a binary-tree augmented with additional links to enable multiple arbitrary sized reductions to run in a non-blocking manner.

Forwarding Links at Intermediate Levels. Figure 5.5(b) shows a binary tree with additional links for forwarding the adder outputs to other adders at the same level, instead of to the parent. This removes contention from three out of the four links, and only one link is shared by Neuron 2 and Neuron 3.

Chubby Links from Root. Physical bandwidth limitations at the root node can limit

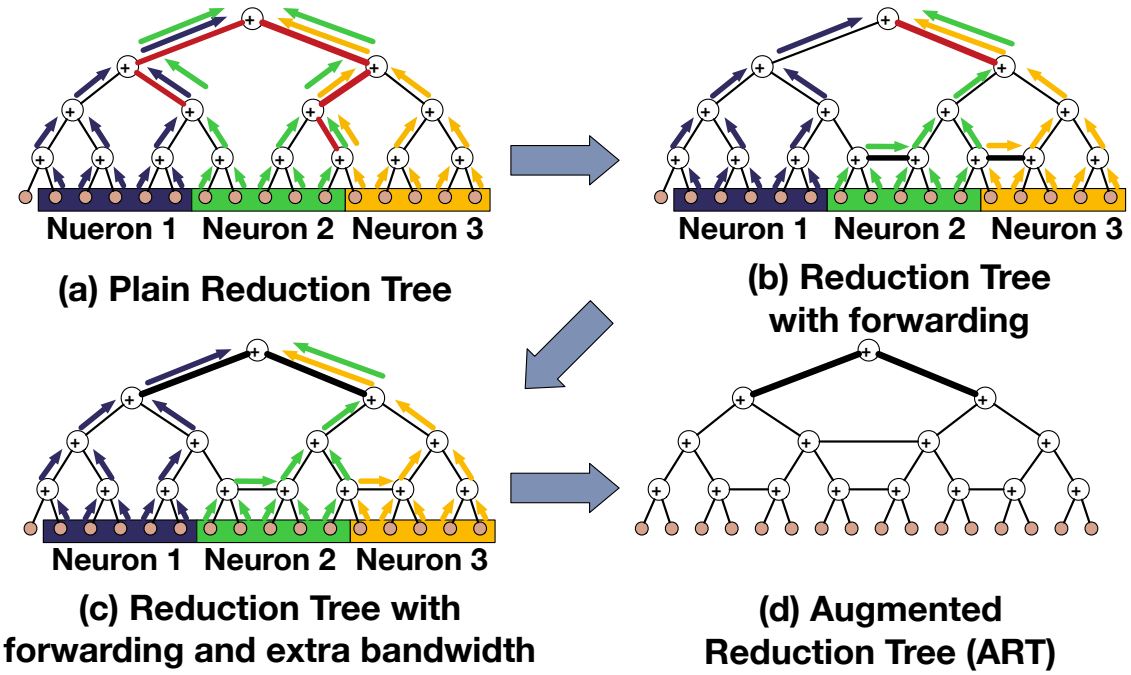


Figure 5.5: Motivation for Augmented Reduction Tree (ART). Three neurons are mapped over five multipliers each. Each neuron generates one output using the adder tree for reduction. Red links in (a) and (b) represent congested links, thick links in (c) and (d) represent links with double bandwidth. The forward links (FL) in the ART are bi-directional.

the number of parallel reductions (collects) as Figure 5.5(b) showed. To address this, we augment the ART with chubby links we discussed in Section 5.2.1. This eliminates contention completely, as shown in Figure 5.5(c).

Properties of ART

Figure 5.5(d) presents an example of an ART. We formally define it and present two key properties.

Definition. *Augmented Reduction Tree is an undirected graph that consists of a complete binary tree and additional links that connects adjacent tree nodes in the same level with different parents except between leaves.*

Property 1: Configurability. *An ART with N leaves can map any adder tree onto its substructure when the adder tree accumulates values from k consecutive leaves and k ¡ N.*

Property 2: Non-Blocking. *An ART can map multiple of such adder trees mentioned in (1) without any sharing any link if the sets of leaves of each adder tree are all disjoint.*

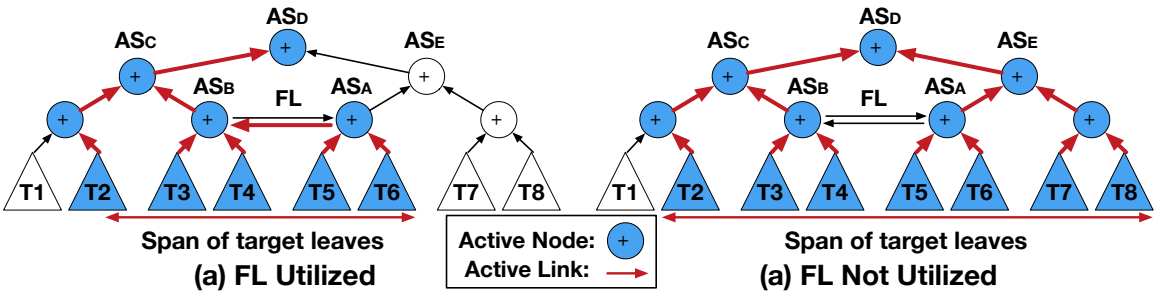


Figure 5.6: Two examples of forwarding link reconfiguration.

Property 1 guarantees any sized reduction operation can be mapped over an ART. Property 2 guarantees that multiple non-blocking reduction operations can be mapped over an ART. For example, if an ART has 32 leaves and we map multiple adder trees that accumulates five values, ART can accommodate four adder trees at the same time providing non-blocking reduction.

Intuitively, additional connectivity between adjacent adder switches on the same level, but with different parents, provides opportunity to accumulate more partial sums closer to the lower levels, which reduces the number of required links in upper levels. Without the additional connectivity, partial sums from the adjacent nodes have to traverse upper levels to be accumulated, increasing the possibility of link congestion. We do not add forwarding links (FLs) between nodes sharing the same parent node because the parent node anyway needs to be traversed to reach the top.

Microarchitecture, Routing, and Flow-Control

The ART is built using *adder switches (AS)*, whose microarchitecture is shown in Figure 5.2. Each AS is *statically configured* to act as either 2:1 ADD, 3:1 ADD, 1:1 ADD plus 1:1 forward, or 2:2 forward. Section 5.3.1 describes the configuration algorithm. Each AS also houses a comparator for POOL layers.

5.3 Mapping Dataflows over MAERI

Our dataflow mapping is mapping neurons one by one over the MSes. We call this Virtual Neuron (VN) Construction. It essentially means configuring the ART, since that is the one that decides which multiplier outputs need to be reduced.

Once the ART is configured, each VN can operate in parallel. The dataflow flexibility in MAERI comes from allowing each VN, which is a MAC operation, to take arbitrary number of MSes (rather than relying on fixed clusters). We also support folding of a VN over itself and multiplex multiple multiply operations over fewer MSes (Section 5.3.8).

5.3.1 Virtual Neuron (VN) Construction over ART

We describe the algorithm for VN construction over the ART for the example in Figure 5.6(a). Each triangle represents a sub-tree, and each circle is an AS. We focus on the FL marked in the figure. Here, the VN spans from T2 to T6 (Figure 5.6(a)).

Step 1: Compute the span of the neuron on the left and right side of the FL. Set the direction of the FL from the smaller to the larger span. We define span to be the number of sub-trees that the VN (generating the psums) crosses. In Figure 5.6 (a), the VN spans from T2 to T4 (i.e., three) on the left, and from T5 to T6 (i.e., two) on the right. The direction of the FL is set from right to left. If the spans are same, then the direction can be set arbitrarily.

Step 2: Check if the sub-trees in direction of the smaller span need to use the parent to this FL on that direction. If not, activate this FL. The parent of FL on the right side (i.e., P_R) does not need to be activated for this neuron, since T7 and T8 are not part of the span. Thus the FL is activated by configuring AS_A to forward the output from T5 to AS_B, and AS_B to act as a 3:1 adder.

Figure 5.6 (b) shows an alternate scenario. Here, Step 1 determines the direction to be left to right. However, on the left, the AS_C has to be activated regardless of the FL, because

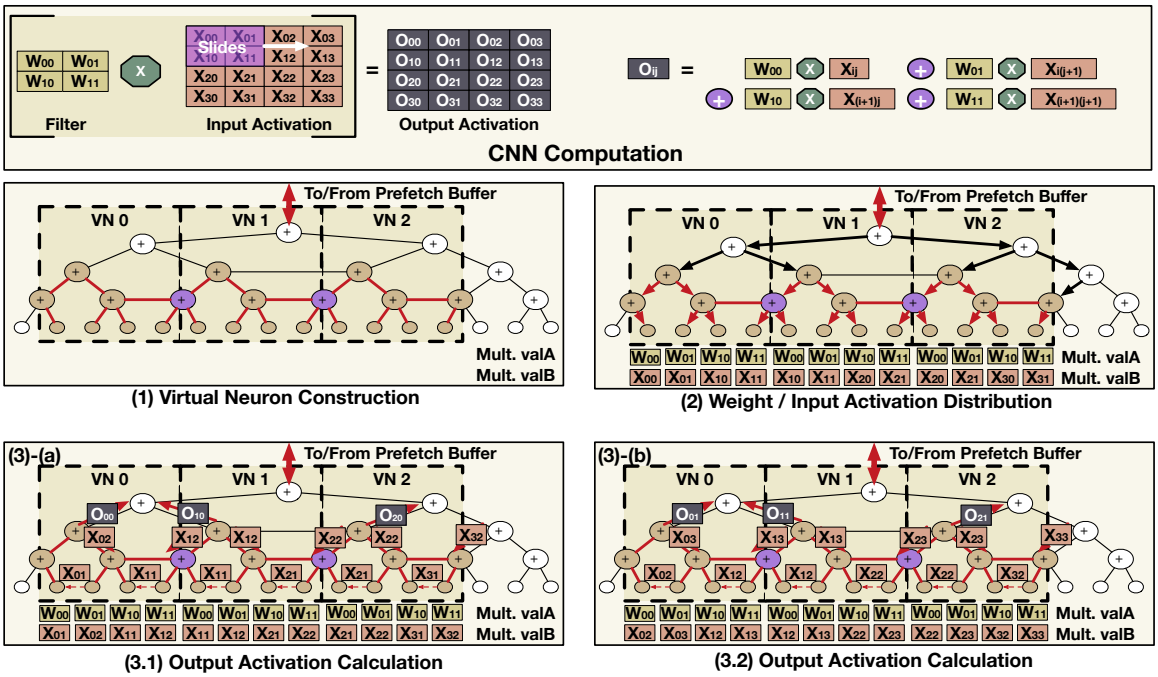


Figure 5.7: CONV2D computation in MAERI. **W**, **X**, and **O** represent weights, input activations, and output activation, respectively.

T2's output goes to it, hence FL does not need to be activated. AS_B is configured to forward its output to AS_C, not AS_A.

How is the span computed by the algorithm? We represent the leaves (MSes) spanning each neuron using a bit-vector. The ART controller starts from FLs in the lowest level to activate FLs, before going up the levels. The number of bits that are 1 on the left and right of this FL in the bit-vector are used to compute the span. Whenever a FL is activated, the bits corresponding to smaller span (i.e., the leaves that will create the psums that will cross this FL) are cleared. This prevents activating multiple FLs in different levels of the ART for the same partial sums.

Next, we demonstrate how example DNN mapping can be mapped over MAERI.

5.3.2 Mapping a CONV2D Layer

We demonstrate how a CONV2D layer can be mapped over MAERI with a walk-through example in Figure 5.7. The weight filter is 2x2, and the input/output activations with one channel are 4x4. This example assumes that each MS stores one input activation and one

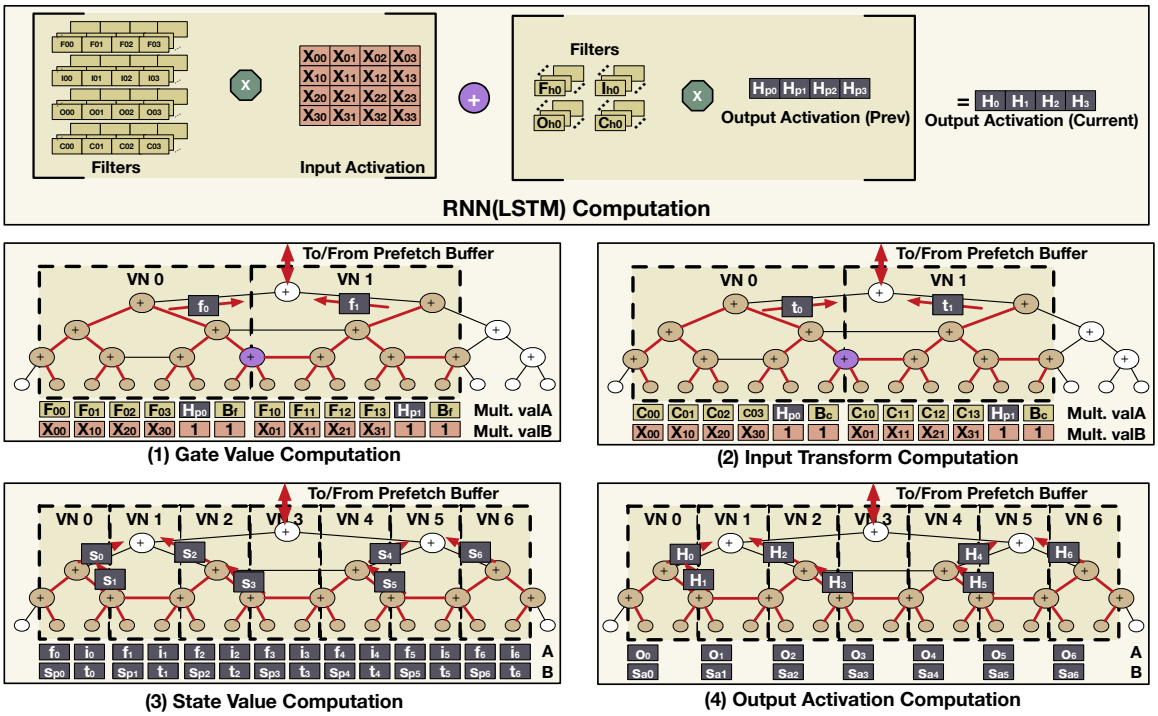


Figure 5.8: LSTM computation in MAERI. **F**, **I**, **O**, and **C** indicate weights for forget/intput/output gated and input transform multiplied with input activations. $\mathbf{F}_h$, $\mathbf{I}_h$, $\mathbf{O}_h$, and $\mathbf{C}_h$ represent weights for forget/intput/output gated and input transform multiplied with previous output activations. **X** and **H** represent input and output activations. The indices of **F**, **I**, **O**, and **C** indicate the ID of corresponding neuron and position within the weight vector (e.g., $\mathbf{F}_{30}$ indicates the first forget gate filter weight value for neuron 4.) The index of $\mathbf{F}_h$, $\mathbf{I}_h$, $\mathbf{O}_h$, and $\mathbf{C}_h$ means its corresponding neuron ID (e.g., $\mathbf{C}_{h3}$ represents the filter weight value to be multiplied with the previous output activation in neuron 4). The four steps presented generate an output activation for each VN. $\mathbf{f}_k$, $\mathbf{i}_k$, $\mathbf{o}_k$, and $\mathbf{t}_k$ represent forget/input/output gate values and input transform at the current time epoch. $\mathbf{B}_f$, $\mathbf{B}_i$, $\mathbf{B}_o$, and $\mathbf{B}_c$ are bias values for each gate value and input transform. $\mathbf{s}_k$ and $\mathbf{s}_{pk}$ are the state values for the current and previous epoch, respectively.

filter weight value locally and the chubby ART, together with the PB, provides sufficient bandwidth to cover all simultaneous reduction flows.

Stage 1: VN Construction.. MAERI first constructs a VN by configuring the ART based on the dimension of the target CNN layer, as Figure 5.7 (1) shows. The controller then maps the filter weights to a set of consecutive multiplication switches in each VN, and configures the corresponding sub adder tree of the ART using the reconfiguration algorithm described earlier in Section 5.3.1. Each VN is responsible for generating one row of output activations, and two VNs can share one AS. In the example, VN 0 and 1 share the AS marked in purple. Although an AS (or a node of the ART) is shared, the overall structure still maintains the non-blocking feature since one of the inputs is sent up the tree, and the other is sent laterally

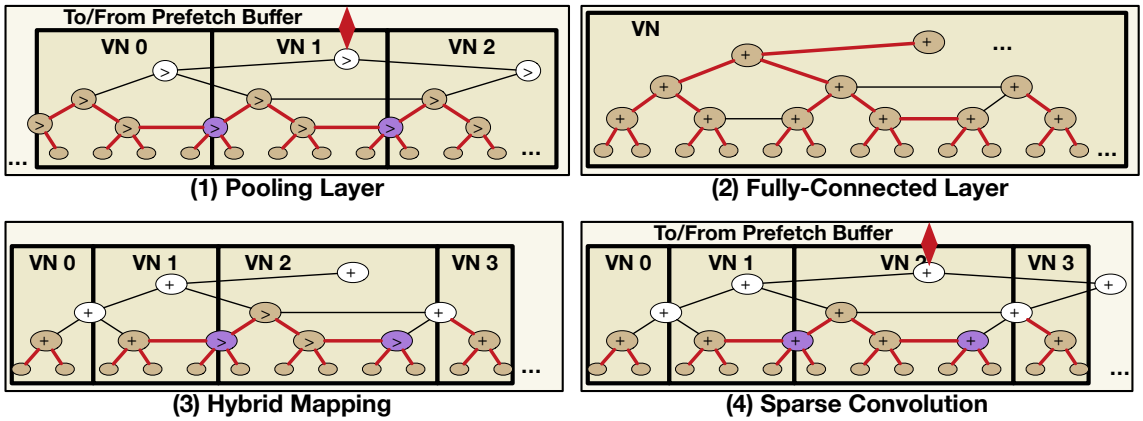


Figure 5.9: Mapping POOL, FC, Cross-Layer and Sparse CNNs over MAERI.

using the forwarding link simultaneously, as Section 5.3.1 described.

This configuration step happens before running each CNN layer and remains constant throughout the run. In case the layer does not fit, it can be folded and mapped multiple times as we explain later in this section.

Stage 2.1: Weight Distribution.. Next, MAERI starts to distribute filter weights, as Figure 5.7(2) shows. Recall that the weight tensor slides over input images in CNNs; as a result the same weight value is required by multiple VNs, each of which is computing an output activation. We exploit the multicast functionality of the distribution tree, by sending one value from the PB and replicating it at the intermediate simple-switches. For example, weight W_{00}, W_{01}, W_{10}, and W_{11} are sent to the first, the second, the third, and the fourth multiplier switch in each VN, respectively. We can exploit the bandwidth of the chubby tree structure to deliver multiple unique weights simultaneously to different multiplier switches. Because each VN requires the same set of weight values, the PB distributes weights only once for every CONV2D layer, which remain stationary throughout the run of the layer.

Stage 2.2: Input Activation Distribution.. The same input activations are used by multiple neurons, just like the weights. For example, both VN 0 and 1 require input activations X_{10} and X_{11}, and both VN 1 and 2 require X_{20} and X_{21}. These are multicasted from the PB. Unlike weight distribution, the distribution of new input activations from the PB needs to be performed whenever each VN has finished computing all psums for one output activation. This part is overlapped (pipelined) with the generation of output activations (Stage 3). For

instance, in Figure 5.7)(3.1), we can see that new input activations X_{02} and X_{12} arrive at VN 0, X_{12} and X_{22} arrive at VN 1, and so on, while it is computing O_{00}.

We model the sliding window behavior of the CNN filter weight tensor by using the local forwarding links between the multiplier switches. Each input activation is forwarded left up to the width of the filter row (which is two in this example) and then discarded. For instance, in Figure 5.7 (3.2), we can see that new input activations X_{01} and X_{11} are forwarded from the second and fourth MSes to the first and third respectively. Because of the data forwarding, each VN requires only two new input activation values (same as row size of the filter) from the PB every cycle, for generating a new output activation. This reduces the bandwidth requirement of the distribution tree, and the overall energy consumption by reducing the interconnect traversal length.

Stage 3: Output Generation.. After the series of initialization steps (VN construction and weight/input activation distribution) finishes, MAERI starts to produce output activation values. Each VN generates output activation values for one row of the output tensor.For the CNN computation example in Figure 5.7, Figure 5.7 (3) shows VN 0 producing O_{00}, followed by O_{01}, and so on. Similarly, VN 1 generates O_{10}, O_{11}, O_{12}, and O_{13}. After finishing one row, input activations corresponding to another row are mapped on the VN, and so on till the end of the current convolution layer.

Optimizing for Spatial Reuse in CNNs.. MAERI tries to optimize and get the best of the three classes of dataflows described in the Eyeriss [**chen2016eyeriss**] taxonomy. Each multiplier switch acts as a *weight stationary* node without requiring weights to be forwarded back and forth. Each row of the weight filter is mapped sequentially across the multipliers of a VN making it *row stationary*. And finally, the configurable ART within each VN acts like an *output stationary* node as it accumulates psums locally. Together, we get a design optimized for high-throughput and low-energy.

5.3.3 Mapping an RNN/LSTM Layer

Figure 5.8 shows how MAERI runs a LSTM layer. A LSTM computation consists of four steps: calculating (1) gate values, (2) input transform, (3) next state value using the results from step 1 and 2, and (4) output activation using the results from step 1 to 3.

Gate values and input transform calculation.. For step 1 and 2, MAERI first constructs VNs. MAERI distributes input activations and weights, and performs the same multiply-accumulate computation as the CONV example. However, unlike the CONV case, it iterates four weight filters (forget/input/output gate and input transform) and reuses input activation values for each gate value and input transform computation. Step 1 and 2 require the same number of MSes within a neuron and share the same input activation data set; thus we merge them in MAERI. In other words, when each VN receives input activations, it reuses them to calculate all the gate values (step 1) and input transforms (step 2) for the received input activation, before the PB distributes the next round of input activation. The computed gate values are collected by the PB (over the ART) and stored.

State value and output activation calculation.. After the completion of step 1 and 2 over target input activations (**X**), MAERI reconstructs VNs to calculate state and output activation values, as Figure 5.8 (3) and (4) show. The reason for reconstructing VNs is because the calculations in step 3 and 4 requires fewer number of multiplier switches; Retaining the same VN configurations as step 1 and 2 would lead to underutilization of the available multipliers. However, sometimes reconstructing VNs between step 3 and 4 may not help if the VNs are too fine-grained (say each VN just has two multiplier switches), since the ART might not have enough bandwidth to compute and transmit all output activations to the root every cycle. Our optimization tool considers this aspect and generates appropriate parameters for the ART controller so MAERI can prevents such a contention scenario.

Step 3 and 4 also calculates partial sums and accumulates them like all the other CNN/RNN computation steps. For state value computation (step 3), each VN receives the previous state value calculated in the previous time epoch from the PB using the distribution

tree, and the forget/input gate values and input transform calculated in step 1 and 2 in the current time epoch. It then generates the current state values based on the received values, as Figure 5.8 (3) shows. For the output activation computation (step 4), each VN receives the output gate value and current state value, multiplies the two, and sends the result over to the activation units to produce the final output activation.

5.3.4 Mapping a POOL Layer

Mapping a POOL layer over MAERI requires creating a VN with with the values to be pooled, and configuring the AS to act as a comparator, rather than an adder. The output of the ART is then the pooled value.

5.3.5 Mapping a FC Layer

A FC neuron gets inputs from all neurons in the previous layer. Correspondingly, the VN for this can be mapped as before, except that it would span many more MSes. In the extreme case, the entire ART can be configured to compute the output for one neuron, as Figure 5.9 shows. In case one neuron does not fit over the free MSes (either because the number of inputs is greater than the total MSes, or some of the MSes are already configured into other VNs), the neuron can be mapped via folding, as Section 5.3.8 discusses.

5.3.6 Mapping Cross-Layers

A cross-layer mapping [107, 71, 72] can easily be supported over MAERI since each VN can be independently configured. Thus each VN could correspond to neurons of the same layer (as Figure 5.7) illustrated, or different layers (Figure 5.9), without requiring any change in the algorithm. In the latter case, the intermediate output from the PB need not be sent to DRAM, but can be streamed to the next VN on-chip directly.

Table 5.1: MAERI implementation details and comparison with Eyeriss and Systolic Array. SysArray/MAERI (Comp) and SysArray/MAERI (Area) are Systolic array/MAERI implementations that have the same number of compute units and area as Eyeriss, respectively.

Design	**Eyeriss**	**SysArray (Comp)**	**SysArray (Area)**	**MAERI (Comp)**	**MAERI (Area)**
Technology	28nm	28nm	28nm	28nm	28nm
Number of PEs (MultSwitches)	168	168	1192	168	374
Local SRAM/PE	512B	0	0	512B	512B
Prefetch Buffer	108 KB	80KB	80KB	80KB	80KB
Area	$6mm^2$	$2.62mm^2$	$6mm^2$	$3.84mm^2$	$6mm^2$

5.3.7 Mapping Sparse Networks

Mapping sparse CNNs is also quite trivial in MAERI, as Figure 5.9 shows. The size of each VN would be different size since the filter sizes vary depending on weight sparsity. Note that MAERI can support mapping of sparse CNNs but might still need additional support to identify sparsity and stored compressed data [74, 20].

5.3.8 Optimization: Folding over Rows

There can be multiple reasons to fold a physical neuron over fewer MSes than its inputs, such as: (i) there are insufficient MSes, or (ii) the bandwidth of the PB or the chubby distribution tree is insufficient to cover all the input activation values during output activation calculation, (iii) the VN is memory bound and waiting for data from DRAM. To support N-way folding, MSes need to have at least N local buffers. This increases their area and power, but lowers the bandwidth requirement from the network.

For example, in Figure 5.7, we can allocate two multiplier switches rather than four within a virtual neuron. In such a mapping, the output of the VN through the ART is an intermediate psum (one row of the filter in this example). This is sent to the PB for temporary storage and then sent back to the corresponding VN to be accumulated into the final output activation that is sent to the activation units.

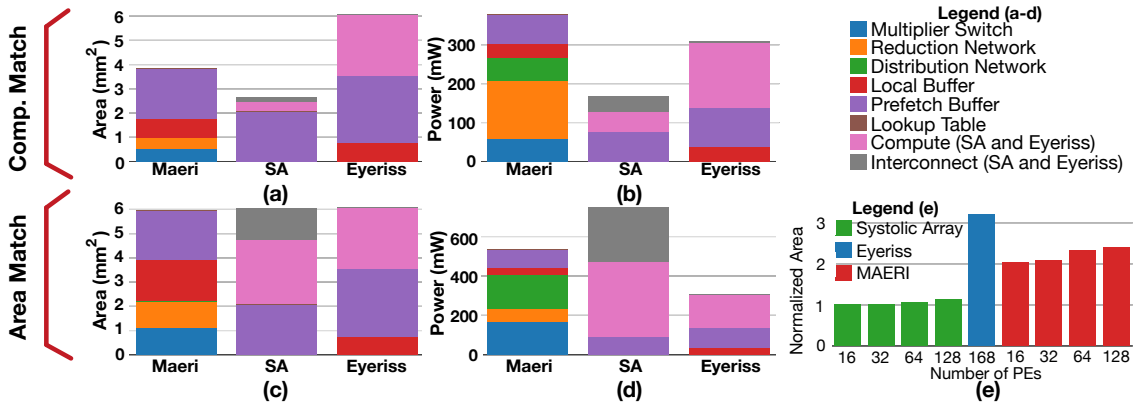


Figure 5.10: Area and power breakdown of MAERI, systolic array and Eyeriss. Comp match (a,b) and area match (c,d) indicate design points with the same number of compute units and area as Eyeriss (Table 5.1). The left and right column plots area (a, c) and power (b, d), respectively. (e) plots the post place-and-routed area of MAERI, systolic array and Eyeriss, normalized to the 16 PE systolic array.

5.4 Implementation

We implemented MAERI in BSV (Bluespec System Verilog) [106] and synthesized it with TSMC 28nm standard cell and SRAM library at 200MHz. For comparison, we also synthesized and placed-and-routed Eyeriss [63][2] and a systolic array [17] in our 28nm environment. We created two design points - one where all three accelerators have the same number of compute (i.e., multiply-accumulate or MAC) units and SRAM size, and one where all three have the same chip area, as Table Table 5.1 shows. We find MAERI to be more area-efficient than Eyeriss for two reasons: (i) its fine-grained MS and AS together are much more area-efficient than a full PE, and (ii) MAERI does not require fully-addressable local register file like Eyeriss; it uses FIFOs instead and relies on delivery of the correct data in the correct order via the distribution tree or local forwarding links. For the same area, MAERI and systolic array can house 206 (2.23 $\times$) and 1024 (7.09 $\times$) more compute units than Eyeriss.

Because of the larger number of compute units and MAERI's near 100% utilization based on its fully non-blocking trees, synthesis tools report higher power in MAERI than Eyeriss. Thinning the adder tree links can bring us to the same power point, while still

[2]We thank the authors of Eyeriss for sharing their RTL with us.

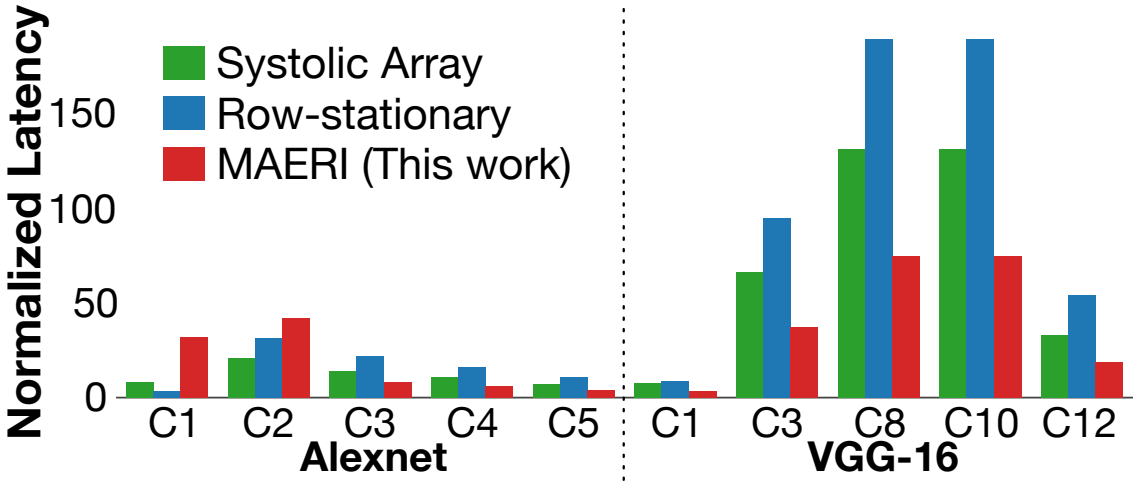


Figure 5.11: Total latency and compute unit utilization of systolic array(SA), Eyeriss [16] style row-stationary accelerator, and MAERI with 64 PEs (multiplier switches) for selected conv layers in Alexnet and VGG16. The latency is normalized to the first Alexnet convolutional layer delay in an ideal accelerator with 64 PEs, infinite bandwidth between all the PEs and the PB, and 1-cycle fixed point computational units.

offering full configurability but at a loss of full utilization every cycle, which might be an acceptable trade-off as a lot of DNN layers are memory-bound [17]. The prefetch buffer (SRAM) dominates in both area and power in the two designs. For the same number of PEs (compute units), the systolic array required the smallest area and power because of its simple structure (MACs connected in a grid). This is also illustrated in Figure Figure 4.7(e). However, systolic array suffers from low utilization [17] and requires a large number of SRAM reads because of the lack of data reuse inside a PE array as we demonstrate later in Section 5.5.3.

5.5 Evaluations

5.5.1 Performance with regular (dense) dataflow.

Given the same number of compute units, the performance of a spatial accelerator depends on both the utilization of the compute units, and the internal dataflow (which determines reuse and activity). We plot the total latency for running selected convolutional layers in AlexNet and VGGnet in Figure 5.11 across the following designs: MAERI, Systolic Array and Row-Stationary (i.e., based on Eyeriss [16]) with 64 multipliers/MACs/PEs respectively. MAERI

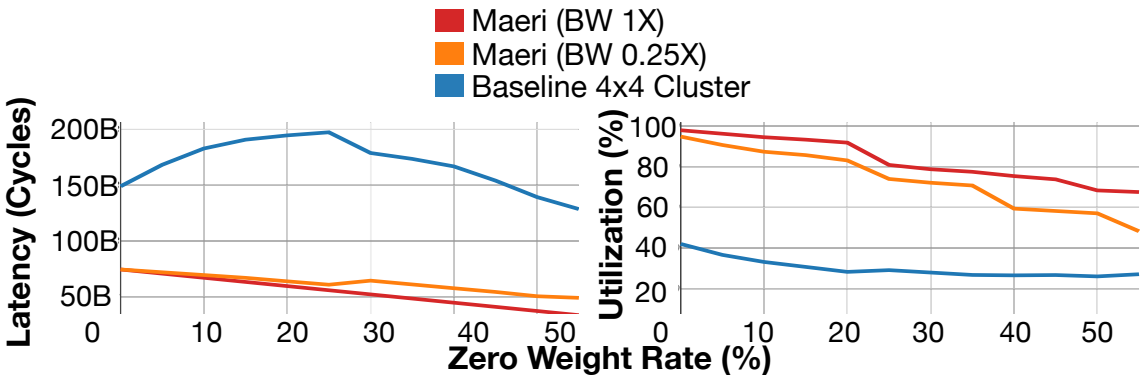


Figure 5.12: Total latency for VGG16 convolutional layer 8 (C8 in Figure 5.11) for sparse workload. MAERI includes 64 MSes. The baseline uses four 4x4 PE clusters connected by buses.

provides a speedup of 72.4% on average across all layers. We observed 95% utilization in average across the multipliers in MAERI. We can see that large filter sizes, such as AlexNet's C1 (11×11; requires temporal folding) and C2 (5×5) layers are adversarial for MAERI as they lead to large VNs, leading to underutilization of the remainder MSes. But recent CNNs like VGG-16 with 3×3 filters provide the best utilization by allowing seven filters to be mapped simultaneously over the 64 MSes with only one MS idle.

5.5.2 Performance with irregular dataflow.

Sparse Dataflow. Figure 5.12 represents the total latency of VGG16 convolutional layer 8 with varying percentage of zero weights executed on MAERI with 64 multiplier switches and different chubby tree bandwidths (1X and 0.25X represent the bandwidth at the root of the tree, i.e., the non-blocking factor). The baseline is modeled similar to SCNN [20] and uses fixed 4x4 PE sized clusters. Even when the workload is dense (i.e., percentage of zero weights = 0), MAERI provides better utilization. This is because VGGNet uses 3x3 filters (and 3 channels). So each OFMAP requires 3x3x3=27 MACs, which use exactly 27 Multipliers/Adders in MAERI, and (4x4)x2=32 MAC units in the baseline, lowering utilization.

When the workload is sparser, the bandwidth requirement for partial sum collection increases because the PE array (multiplier switches and ART in MAERI) can cover more number of neurons at the same time. This becomes a bottleneck for the baseline where

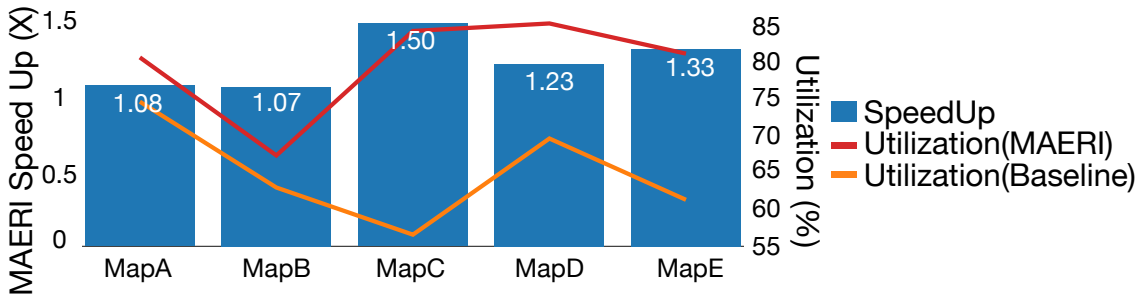


Figure 5.13: Speed up of MAERI over a 64 PE baseline (four 4x4 clusters) with hybrid cross-layer dataflow. MapA-E are Alexnet conv1+2+3, 2+3+4, 3+4+5, 1+2+3+4, and 2+3+4+5 respectively.

the clusters are connected by a bus (which limits bandwidth) even though the total number of computations goes down in a sparse workload. In contrast, MAERI provides 73.8% utilization even at 50% sparsity, and correspondingly 6.9 × speedup.

Cross-Layer Dataflow. We model five cross-layer dataflows by fusing a combination of AlexNet convolution layers. Figure 5.13 plots the utilization, and speedup of MAERI over a baseline accelerator with four 4x4 clusters (i.e., a 16:1 reduction trees in each). We observe 1.08 - 1.5 × speedup. Accelerators based on fixed clusters or fixed sized reduction trees are inefficient in utilizing all the PEs for certain mappings. For example, in Map C, we map three convolutional layers with three 3×3 filters. In the baseline, we can only use 9 PEs within each cluster; trying to utilize the remaining 7 PEs introduces non-uniform traffic that fixed interconnection cannot support. However, MAERI with its flexible interconnects can support such irregularity and thus maintains high utilization and provides maximum speedup.

5.5.3 MAERI Deep Dive

ART vs Fat Tree vs Plain Tree. Figure 5.14 plots the utilization of the MSes across three kinds of reduction trees: ART, fat-tree, and four plain adder trees as the size of each virtual neuron (VN) mapped over the trees increases. Our aim is to quantify the benefits discussed earlier in Figure 5.5. We can observe that the ART provides relatively uniform and high utilization while fat tree and plain adder trees involves significant fluctuation in utilization. Such fluctuations occur because the efficiency of plain adder trees and fat trees are highly

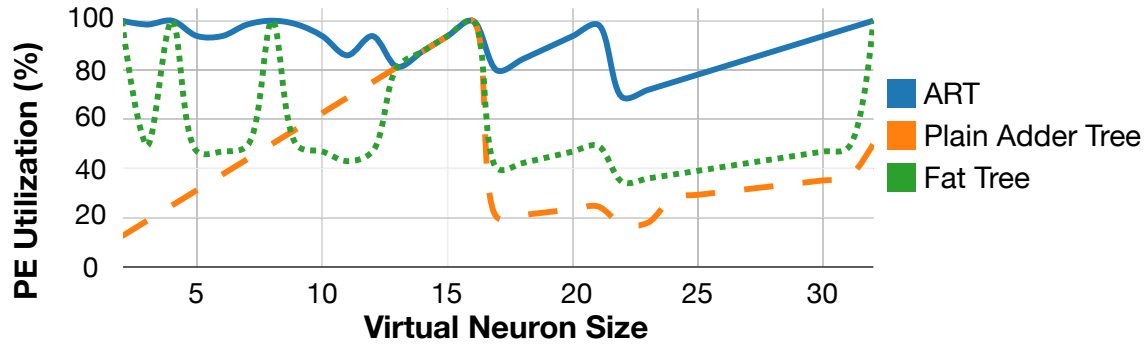


Figure 5.14: PE utilization with ART, fat-tree and four 16-wide plain adder trees when 64 PEs are used.

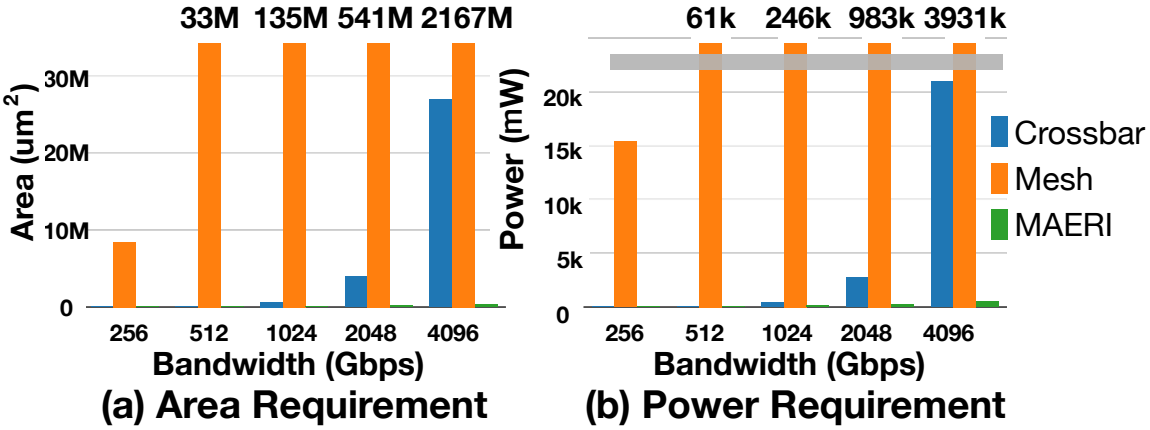


Figure 5.15: Area and power comparison between the NoC in MAERI and traditional NoCs.

sensitive to VN size (number of non-zero weights within a filter of a channel). The plain adder trees have low utilization because VNs less than size 16 end up instantiating an entire tree with idle multipliers, and thus provide 100% utilization only at a VN size of 16. If the VN size is a power of 2, the Fat Tree works identical to the ART since all operations fit within the binary tree structure and the ART's forwarding links are not required. When the VN size is not a power of two (which is the case in VGGnet and in sparse designs), the utilization of a fat-tree drops but ART continues to provide high utilization. ART also has fluctuations of utilization in cases where the total multipliers (64 in this example) is not a multiplier of the VN size as that leads to idle multiplier switches due to temporal folding (see Figure 5.16 (b)). Support from an advanced compiler may enable utilizing such multipliers by temporally folding large VNs over few multipliers.

MAERI trees vs. traditional NoCs. Compared to other traditional NoC designs, the

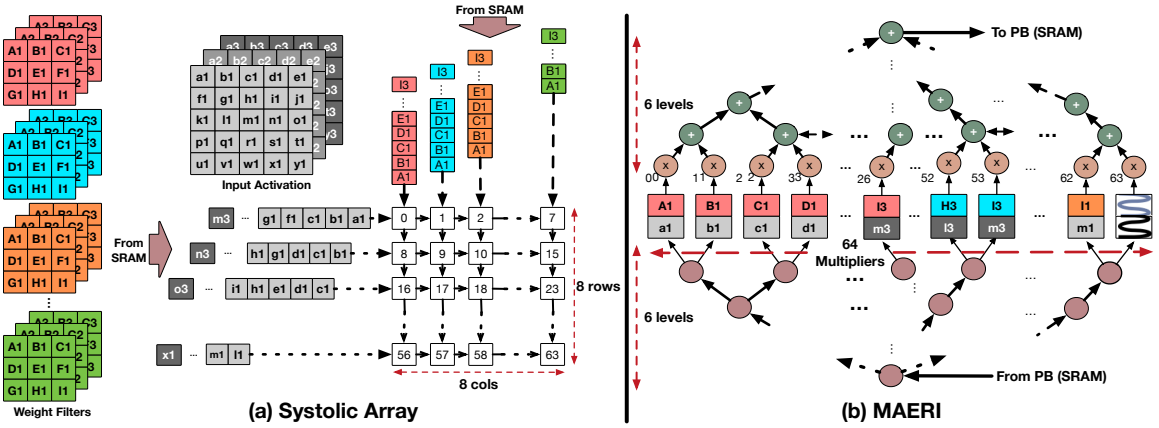


Figure 5.16: A mapping example of a convolution layer with eight 3x3x3 filters and 5x5x3 input activation over (a) a systolic array with 64 PEs and (b) MAERI with 64 multiplier switches.

NoC in MAERI is highly area- and power-efficient, as post-synthesis area and power overhead over NoC bandwidth plot in Figure 5.15 presents. The overhead of the NoC design in MAERI is minimal compared to mesh or crossbar while the NoC design provides sufficient bandwidth for the traffic in MAERI. This is because the NoC architecture in MAERI is optimized just for communication patterns within DNN accelerators, and it consists of extremely light-weight building blocks.

MAERI vs. systolic arrays. Systolic array is also considered as a major approach [17] to implement an accelerator because of the low PPA cost based on its relatively simple design and high parallelism it provides. We compare a systolic array-based design and MAERI using the example presented in Figure 5.16, which is a convolutional layer with eight 3x3x3 weight values, 5x5x3 input activation values, and a stride of one. In the example in Figure 5.16, both the systolic array (a) and MAERI (b) contain 64 PEs (i.e., MACs in Systolic array, multiplier and adder switches in MAERI). The systolic array reads input activation and weights from the left and the top of the array, respectively. Because systolic array is based on store-and-forward dataflow style, a controller needs to carefully adjust the injection time of each data, which lets the PEs remove complicated hardware for control data I/O. For example, if weight value A1 in the first 3D filter (red) and input activation a1 are injected to PE 0 at cycle *t*, weight value A1 in the second 3D filter (blue) needs to be injected at cycle *t+1* because input activation a1 arrives at the PE 1 at cycle *t+1*. To

process one sliding window in the example convolution layer, the systolic array needs to read 216 weights and input activation (3×3×3×8). Because each sliding window is mapped on each row of the systolic array, the systolic array can process eight sliding windows within an iteration. Each iteration requires not 27 (the number of partial sums to be generated in a sliding window) cycles but 43 cycles (27 + 8 (injection delay)+ 8 (time to process the last weight/input activation set, which are weight I3 in the last filter (green) and input activation y3)) and generates eight output activations. Because the example requires sliding the window 25 times, the total number of iteration is four with an incomplete iteration that computes only one output activation at the last iteration. Therefore, the total cycles to process the example layer is 156 cycles (43 × 3 + 27). Although systolic array provides high throughput, it cannot reuse data within the PE array so it requires to read different data every cycle to each row and column, which results in high energy consumption [16]. Therefore, the systolic array need to read 1,323 times from the SRAM in the example.

In contrast, MAERI minimizes the number of SRAM reads by weight reuse in multiplier switches and multicasting input activations, which requires 516 reads (35% compared to the systolic array) for the same example. To the process the same convolutional layer in the example, MAERI first maps each channel in 3D filters (nine weights) as a VN across the multipliers, creating seven VNs in this example (with the last multiplier idle). We utilize temporary register in each adder switch to accumulate output activations from folded filters mapped in different iterations. In this manner, the number of iterations is four, and each iteration consumes 37 cycles (1 for configuration + 9 for weight distribution + 27 for multicasting input activations) with 8x chubby distribution tree. Therefore, MAERI requires 143 cycles to process the example convolutional layer, which reduces 9% of total latency compared to the example systolic array presented in Figure 5.16 (a).

In summary, MAERI provides 9% better throughput and 65% less SRAM reads in the above example. Extending the same analysis to 256x256 systolic array (TPU [17] specifications) vs MAERI with 256x256 multipliers on VGG16, we observe MAERI issues

```
typedef 16 DistributionBandwidth;
typedef 16 CollectionBandwidth;
typedef 32 NumMultSwitches;
```

<Hardware Resource Description>

```
K 64
C 64
R 3
S 3
Y 224
X 224
```

<Layer Dimension Description>

(a) Example Input Files for RTL Simulation

```
@ Cycle  369885197: Received all the outputs; Testbench terminates
 Layer dimension K =        512, C =        512, R =          3, S =          3, Y=
 14, X =         14
 Output dimension:        512 x         12 x         12

Number of injected weights:    7077888
Number of injected inputs:  396361728
Number of injected unique inputs:     109568
Number of input multicasting:  103809024
Number of generated partial sums: 1019215872
Number of performed Ops (Multiplication and Addition): 2000683008

Total runtime (assuming 1GHz clock):  369885197 ns
```

(b) Example RTL Simulation Output

Figure 5.17: Example input and output of MAERI simulation.

6.3x less memory reads. Furthermore, the improvements can be more significant in irregular dataflows such as sparse and inter-layer fusion. However, the better performance and energy efficiency from less SRAM reads of MAERI comes at an area cost, as Figure 5.10 shows.

5.6 MAERI Codebase and Availability

The source code of MAERI is available via the following link: https://github.com/georgia-tech-synergy-lab/MAERI. General information about MAERI with link to related tutorials is available via the following web site: http://synergy.ece.gatech.edu/tools/maeri/.

MAERI provides RTL simulation via Bluesim [106]. Figure 5.17 shows example input and output of MAERI simulation. As inputs, users need to specify the amount of hardware resources and layer information. As outputs, MAERI simulator reports the total runtime with extra information such as number of multicasts, as shown in Figure 5.17 (b).

5.7 Summary

In this work, we present and evaluate MAERI, an accelerator substrate for mapping arbitrary dataflows that arise in DNNs due to its topology or mappings. Our approach is to augment multipliers and adders with tiny switches, and interconnect them via a novel reconfigurable interconnect that supports arbitrary sized neurons. We demonstrate how MAERI is not only capable of running CONV, LSTM, POOL and FC layers, but also supports cross-layer mapping and sparsity. MAERI's NoCs add minimal overheads over NoCs in state-of-the-art CNN accelerators while providing tremendous flexibility. We believe that MAERI is robust to supporting new optimizations in DNNs as it can construct arbitrary sized MAC engines very efficiently. This work also opens up exciting opportunities in compiler designs that can take arbitrary DNNs and map them efficiently over a MAERI-like fabric [30].

In this chapter and Chapter 4, we proposed tiny-switch-based reconfigurable NoC solutions for enabling mapping flexibility. In the following chapter, we explore another approach to enable mapping flexibility in DNN accelerators via heterogeneity.

CHAPTER 6

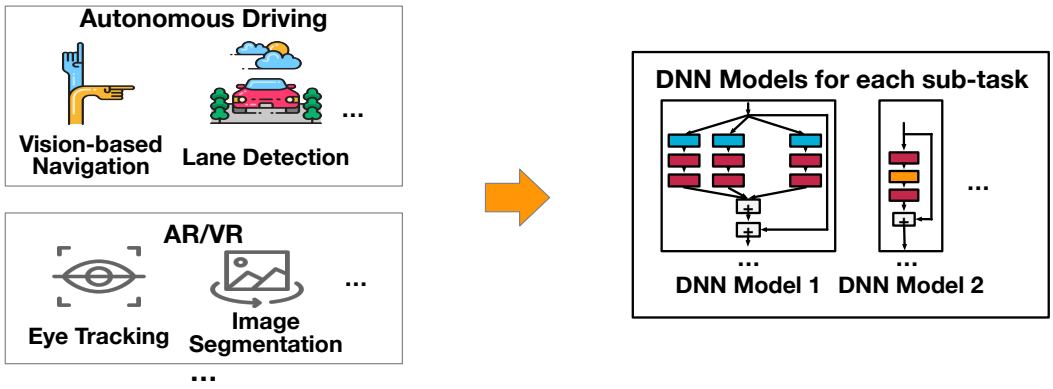
HERALD: COST-BENEFIT ANALYSIS AND AN OPTIMIZATION FRAMEWORK OF HETEROGENEOUS DNN ACCELERATORS

In this chapter, as another approach to enable mapping flexibility in DNN accelerators, we explore heterogeneous DNN accelerators (HDAs). In Section 6.1, we first present an emerging use case of DNN accelerators, real time multi-sub-task applications, and clarify how HDAs can efficiently accelerate such applications. In Section 6.2, we discuss the proposed HDA optimization framework, Herald, that addresses challenges of designing HDAs and maximizes benefits of HDAs. In Section 6.3, we perform case studies to show the efficacy of Herald and the benefits of HDAs.

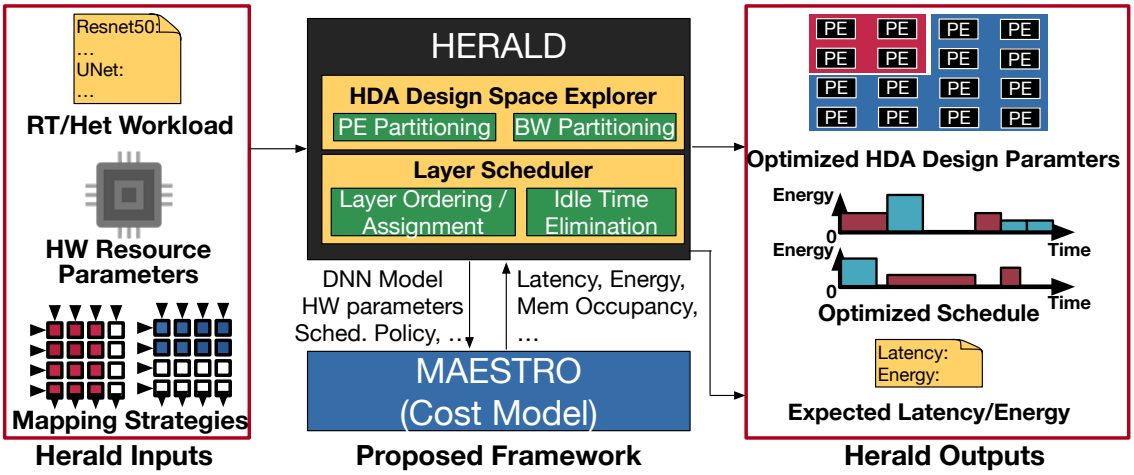
6.1 Motivation

6.1.1 Realtime and Heterogeneous (RT/Het) Workloads

The number of applications relying on DNN inferences are increasing in various domains. Computer vision is one of the most popular domains of such applications, which encompass various tasks like face recognition, image segmentation, image super resolution, depth estimation, and so on [109]. Recent practical use cases such as autonomous cars, AR glasses, and VR headsets often consist of many of such tasks to enable their desired functionalities. Such use cases often result in real-time (i.e., operating with a certain target processing rate) and heterogeneous DNN workloads (i.e., many *different* DNN models and layers with diverse layer operation and size). For example, a VR headset simultaneously runs DNN models for hand tracking, pose estimation, action segmentation, and multiple image classifications [110] at designated frame rate for each. Another example is an autonomous car that performs lane detection [111], driving scene understanding [112], and so on. In this



(a) Realtime and Heterogeneous DNN Workload from Multi-task Realtime Applications



(b) An Overview of Proposed HDA Optimization Framework (Herald)

Figure 6.1: Motivational workloads for HDAs and the HDA optimization framework, Herald, we propose.

thesis, we term the workloads for such use cases as real-time and heterogeneous (RT/Het) DNN workloads.

Real-time and heterogeneous (RT/Het) DNN workloads often target designated processing rate (or, frame rate) based on their functionalities, which imposes more stringent latency constraints to DNN accelerators than before for the quality of service (QoS). Another challenge based on the heterogeneity of the RT/Het DNN workloads with diverse layer operation and layer sizes. Such heterogeneity causes efficiency fluctuation when they run on an accelerator with a mapping style since no single mapping provides good efficiency across all the layer types and sizes [19]. We next discuss the details of the heterogeneity challenge in detail and proposed approach, heterogeneous DNN accelerators (HDAs).

Table 6.1: DNN models selected for case studies motivated by AR/VR workloads [110]. For works without model name, we name them to refer to those works in the rest of paper.

Task	Model	Layer Shape	Layer Operations
Image Classification	Resnet50 [14]	Classification	CONV2D, FC, Skip-Con.
Image Classification	MobileNetV2 [5]	Classification	CONV2D, DWCONV, Skip-Con.
Hand Tracking	UNet [100]	Segmentation	CONV2D, FC,TRCONV, Concat.
Depth Estimation	Focal Length DepthNet [113]	Segmentation	CONV2D, FC, UPCONV
Hand Pose Estimation	Br-Q HandposeNet [114]	Classification	CONV2D, FC

6.1.2 Layer Heterogeneity in DNN Models

Because of the diversity of sub-tasks in compound DNN workloads, the layers in the DNNs are also diverse, or *heterogeneous* while DNN accelerators are often optimized for specific DNN models to exploit the benefits of specialization. Therefore, understanding the layer heterogeneity and optimizing for them is crucial to efficiently support compound DNN workloads.

From recent DNN models, we can observe two classes of heterogeneity; layer shape, which refers to the sizes of each dimension of activation and filter (or kernel), and layer operations. We summarize the two classes of heterogeneity of some recent DNN models related to multi-DNN applications such as autonomous driving and AR/VR devices in Table 6.1.

Layer Shape

Classification networks such as Resnet [14] gradually reduce the resolution of activation because their goal is to extract a classification vector where each entry represents the probability of each class. Also, classification networks tend to increase the number of channels to exploit as many features as possible for accurate classification. Therefore, layers in classification networks have high-resolution activation and shallow channels in early layers and low-resolution activation and deep channels in late layers, as illustrated in Figure 6.2 (a).

In contrast, segmentation networks such as UNet [100] need to restore the original resolution of activation because their goal is to generate masks over target objects in the input image. However, segmentation networks still need to extract as many features as those

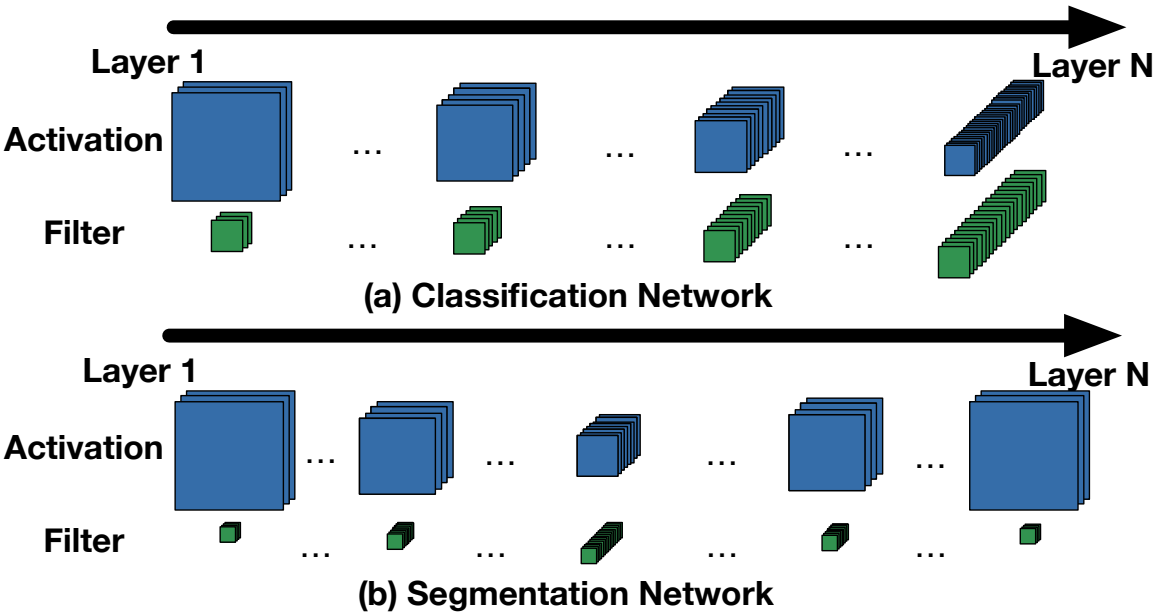


Figure 6.2: Trends in layer shape of (a) classification convolutional neural networks (CNNs) such as Resnet [14] and (b) segmentation CNNs such as UNet [100]. Green and blue squares refer to the filter and input/output activation tensors. The size of squares (width, height, and depth) represent the size of each tensor dimensions we discussed in Figure 2.12

in classification networks for high accuracy. Therefore, segmentation networks first follow the same trend as classification networks until the mid-layer. Afterward, segmentation networks reduce the number of channels and gradually restore the resolution of activation using up-scaling methods such as transposed convolution (a.k.a. deconvolution or up-convolution). As a result, layer shapes in segmentation networks follow the trend illustrated in Figure 6.2 (b).

Layer Operation

We list lists DNN models of computer vision tasks related to AR/VR in Table 6.1. As listed in layer operation column of Table 6.1, layer operations in such models are diverse and heterogeneous. Each layer operation prefers different mappings and hardware [19], which makes such workloads challenging to monolithic accelerators.

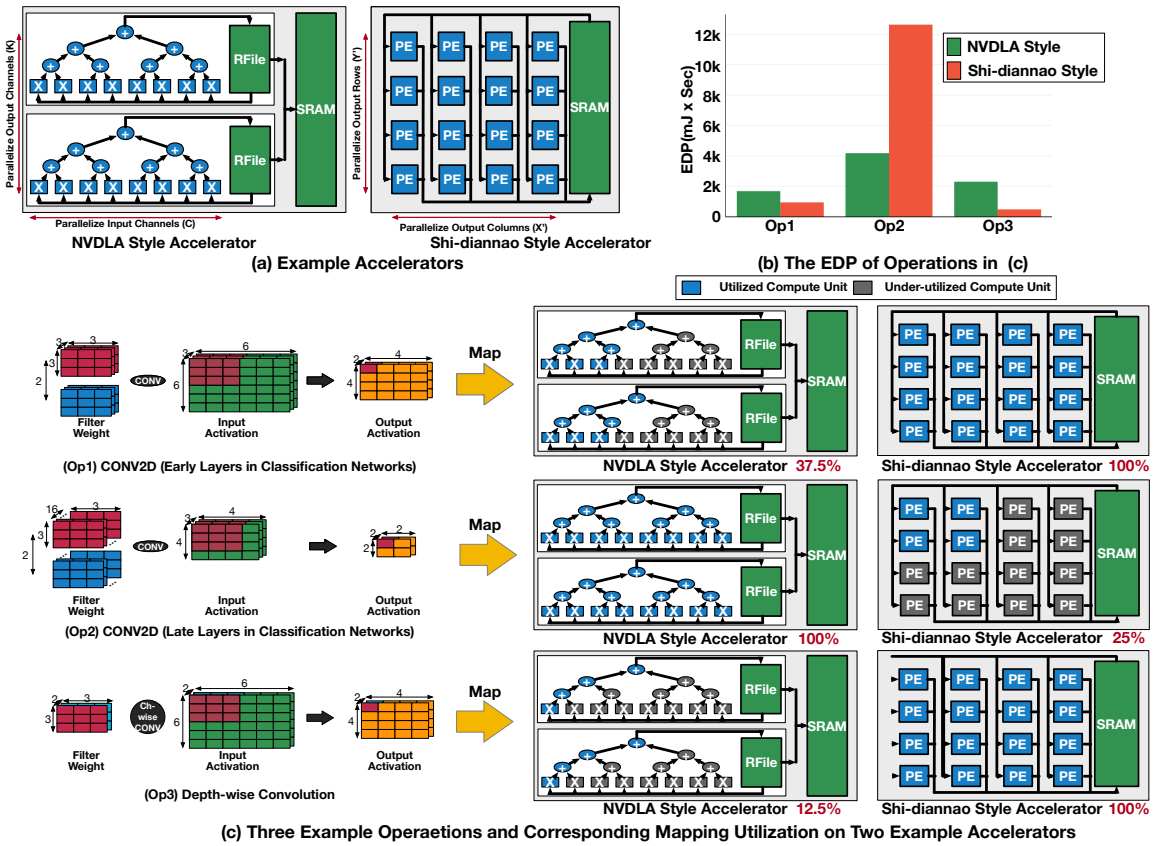


Figure 6.3: The impact of mapping styles on efficiency. (a) Two example accelerators based on NVDLA [49] and Shi-diannao [61] mapping styles. (b) the EDP as the indicator of efficiency (lower is better) of two example accelerators on three example operations presented in (c). (c) three example operations based on CONV2D and depth-wise CONV2D and mapping utilization of compute units on each example accelerator based on their mapping styles. We term the mapping utilization as the number of PEs with computation mapped divided by the number of PEs to distinguish it from the under-utilization based on stalls at execution time due to insufficient network-on-chip (NoC) and memory bandwidth.

6.1.3 Mapping and Efficiency of Accelerators

In Chapter 3, we discussed that no single mapping style is good for all the layers, which implies that we need to optimize mappings for each layer in target DNN workloads to maximize the efficiency of an accelerator. To provide more intuition, we show examples that shows the impact of mappings on efficiency in Figure 6.3, comparing two example accelerators based on Shi-diannao [61] and NVDLA [49] mapping styles. Those two accelerators have distinct approaches to compute MAC operations in DNNs. As illustrated in Figure 6.3 (a), a Shi-diannao style accelerator parallelizes activation width and height using an output-stationary style mapping, which exploits convolutional reuse across kernels,

while an NVDLA style accelerator parallelizes input and output channels using a weight-stationary style mapping, which exploits activation reuse across output channels. Also, Shi-diannao style employs output-stationary style mapping that maximizes output reuse, while NVDLA style employs weight-stationary mapping that maximizes filter weight reuse. Such differences in mappings result in dramatically different utilization of compute units, as shown in Figure 6.3 (c). We use three example operations presented in Figure 6.3 (c) to show the impact of mappings. Op1 and Op2 are CONV2D operations with the aspect ratio of early and late layers in classification network introduced in Figure 6.2 (a), respectively. Op3 is a depth-wise CONV2D operation with the same layer size as Op1.

Based on the parallelization strategies of each example accelerator and layer sizes, we can observe dramatically different PE utilization as shown in Figure 6.2 (c). We use MAESTRO [19] cost model discussed in Chapter 3 for DNN accelerators to estimate the latency and energy and compute energy-delay product (EDP) as one of the indicators of overall efficiency, as shown in Figure 6.2 (b). In combination of the differences in utilization and data reuse strategies, two example accelerators result in dramatically different EDPs, which implies distinct preference of two example accelerators to the operations. In addition to the mapping utilization, each of mapping style has dramatically different memory/network-on-chip(NoC) bandwidth requirements, buffer size requirements, and so on, which also varies based on the layer shape and operations in a different degree [19].

Therefore, we can observe again that no single mapping style is good for all the layers and we need to optimize the mapping for each layer to maximize the efficiency of an accelerator. However, the common DNN accelerator design practice for heterogeneous workloads is to optimize the mapping for the average case of the workload, which can result in a consistently inefficient mapping for all the layers in the workload. This is one of the major challenges for DNN acceleration for emerging RT/Het DNN workloads. Considering the challenge, how do we design an accelerator that can keep the high efficiency even if the workload is heterogeneous?

To cope with the challenge, we propose to design heterogeneous DNN accelerators (HDAs) that contain multiple sub-accelerators with distinct mapping styles within a single accelerator chip, which can provide mapping flexibility. We discuss how HDAs can provide efficiency benefits and what are challenges for HDAs next.

6.1.4 Benefits of HDAs for RT/Het DNN Workloads

HDAs have two potential benefits over monolithic accelerators, which are accelerators running one mapping style, specialized for a specific layer operation and a range of layer size.

Selective scheduling. Because each layer differs by operation and shape prefers different mapping style and hardware, running each layer on its most preferred sub-accelerator in an HDA is an effective solution to maximize overall efficiency.

Latency hiding via layer parallelism. Unlike most of the monolithic accelerators run one layer after another, HDAs can run multiple layers of different models on each sub-accelerator in parallel. By running multiple layers in parallel, a heterogeneous accelerator can overlap the latency of multiple models, which leads to latency hiding among DNN models reducing overall latency.

6.1.5 Challenges of HDAs

Although HDAs can provide potential benefits in micro-specialization for each layer, naive HDA designs and schedulers can lead to inefficiency.

Reduced parallelism within a layer. Given the same number of PEs between a monolithic and an HDA, sub-accelerators in the HDA has smaller number of PEs than the monolithic accelerator since hardware resources need to be distributed (or partitioned) for each sub-accelerator. Therefore, the maximum degree of parallelism each sub-accelerator can exploit for a layer can decrease compared to a monolithic or flexible accelerator. That can lead to higher energy consumption since the amount of data reuse can also decrease (depending on

the mapping) if the degree of parallelism decreases.

Hardware Resource Partitioning. Each sub-accelerator contains smaller amount of hardware resources than a monolithic DNN accelerator if we assign the same hardware budget to the entire chip because we need to distribute hardware resources across sub-accelerators in an HDA. That is, if the hardware resource distribution is sub-optimal, the overall efficiency of an HDA chip can be also degraded.

Scheduling and Memory Constraints. Because multiple sub-accelerators exist in an HDA sharing one global buffer, scheduling of layers that determines sub-accelerator-layer matching and ordering of the execution is now critical for overall efficiency, which is a problem did not exist in monolithic accelerators.

To deal with the challenges of HDAs and exploit benefits of HDAs, we propose Herald, an HDA optimization framework that consists of a design time (hardware resource distribution optimization) and a compile time (layer scheduling) optimization framework, performing (1) design and compile time co-optimization when a user designs a specialized HDA for a given workload or (2) compile time optimization only after an HDA is deployed and the target workload changes. Exploiting those sub-components, Herald automates HDA design tailored for user-specified target models and outputs estimated latency and energy using the co-optimized design. We discuss details of Herald next.

6.2 Herald Framework

In this chapter, we discuss the methodology in the main use-case of Herald that co-optimize hardware partitioning and layer execution schedules at design time. In Section 6.2.1 and Section 6.2.2, we discuss the execution model and latency/energy estimation methodology for each design point. In Section 6.2.3, we discuss the static hardware resource partitioning. In Section 6.2.4, we discuss the layer scheduling on HDAs, which can be run independently at runtime when the workload changes.

6.2.1 Execution Model

We target layer granularity execution on each sub-accelerator of HDAs because we observe significantly different mapping preference of layers [19, 21] and more fine-grained scheduling results in high control and scheduling overhead. We assume the following execution steps of accelerators in Herald.

1. Fetch filter weight values from DRAM and store them in a global buffer.
2. Distribute filter values to sub-accelerators based on layer execution schedule.
3. Fetch activation from DRAM and store them in the global buffer.
4. Stream activation values to their corresponding sub-accelerators based on layer execution schedule.
5. Store streamed-out output activation from each sub-accelerator to the global buffer.
6. During sub-accelerators compute output activation, fetch next filter values from DRAM and send the filter values to the next accelerator (assumes double-buffering).
7. When a sub-accelerator finishes executing a layer, stream output activation stored in the global buffer as input activation of the next layer.
8. Repeat above processes until processing all the layers of all the models.

For steps 3 and 6, activation is stored in DRAM and loaded in a tiled manner specified by the mapping in target accelerator if the buffer size is not sufficient to store entire activation. When output activation is committed to the global buffer, Herald by default assumes a rearrange buffer that adjusts the data layout for the next layer if it runs on another sub-accelerator with a different mapping style. In the evaluation, we select mappings that have the same inner-loop order so that we can maintain the same data layout, which eliminates sub-accelerator context change overheads from different data layout. When the data layout and miscellaneous context change overheads, Herald also provides an option to specify the latency and energy penalties for them.

6.2.2 Latency and Energy Estimation

We extend MAESTRO [19, 115] discussed in Chapter 3 for latency and energy estimation of HDA designs with given schedules. Although MAESTRO supports any mapping, it does not model multi-DNN sub-accelerator environment, which is crucial for HDA evaluations. Therefore, we extend MAESTRO to support multi-DNN accelerator environment with heterogeneity. Herald models the memory requirement for the global buffer and data movement from/to the global buffer to/from sub-accelerator buffers. The modeling method follows the same methodology proposed by MAESTRO, which identifies the amount of reuse and computing activity counts based on them (for energy) and communication/computation delay considering reuse (for latency). In addition to the same analytic equations, Herald considers the layer execution schedule generated by the scheduler we develop, discussed in Section 6.2.4 by modeling non-synchronized execution of sub-accelerators (i.e., each sub-accelerator start processing a layer as soon as input data are available). For estimating latency and energy of sub-accelerator runs, we exploit the original MAESTRO cost model.

6.2.3 Accelerator Design Space Exploration

Herald models accelerators using various hardware parameters such as number of PEs, network-on-chip (NoC) bandwidth, NoC latency, global memroy size, global memory bandwidth, and so on. Herald exposes unit cost database so that users can update unit area/energy costs for each hardware component so that users can easily update it to evaluate HDAs under their own environment (technology node, etc.).

Herald's design space exploration (DSE) tool receives the total number of PEs, memory size, and memory/NoC bandwidth as inputs, which describes the overall hardware budget for an accelerator chip. Unlike monolithic DNN accelerators fully exploit them for a single accelerator substrate, HDAs need to distribute such resources for each sub-accelerator. However, evenly distributing those resources (i.e., naive HW resource partitioning) does not yield the most optimal HDAs because each accelerator's mapping style has a different

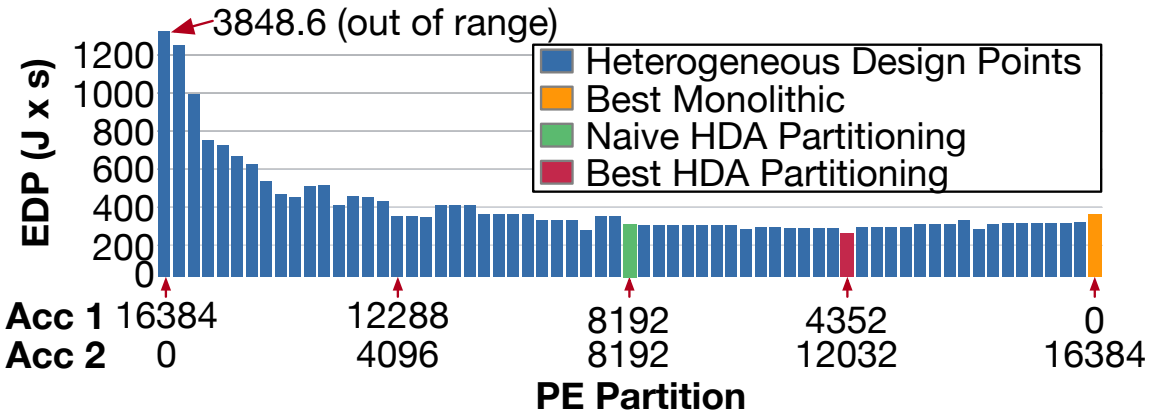


Figure 6.4: The impact of PE partitioning upon a large accelerator listed in Table 6.3 with two sub-accelerators (ACC1: Shi-diannao style, ACC2: NVDLA style). We use evaluation workload A presented in Section 6.3. The left- and right-most represents ACC1 and ACC2 monolithic designs.

optimal balance between the number of PEs, memory size, and bandwidth requirements as discussed in Section 6.1.3. Therefore, Herald explores the resource partitioning space for each resource type, which constructs a nested combinatorial optimization problem, or a nested resource partitioning problem.

We implement a combinatorial optimization framework upon an analytic cost model for DNN accelerators, MAESTRO [19], which estimates the latency and energy for given layer, mapping, and hardware design parameters (number of PEs, NoC bandwidth, memory size, etc.). In Figure 6.4, We show an example design space from PE partitioning over two sub-accelerators in a 16K-PE-HDA fixing other design parameters assuming naive bandwidth partitioning (128/128 GBps). We can observe that the design space has irregular computational cost variations so evenly partitioning (8K/8K PE partitioning) does not yield the most efficient HDA.

Based on user-specified framework options, Herald's design space exploration (DSE) tool either performs an exhaustive search, binary sampling-based search, or random search. Binary sampling-based search first evaluates 2^n design points with a regular interval with user-specified parameter n. Afterward, Herald selects an interval between two adjacent evaluated design points with the lowest average cost and performs an exhaustive search over the selected interval. The random search follows a similar approach as the binary-sampling-based search but selects random pre-evaluated design points.

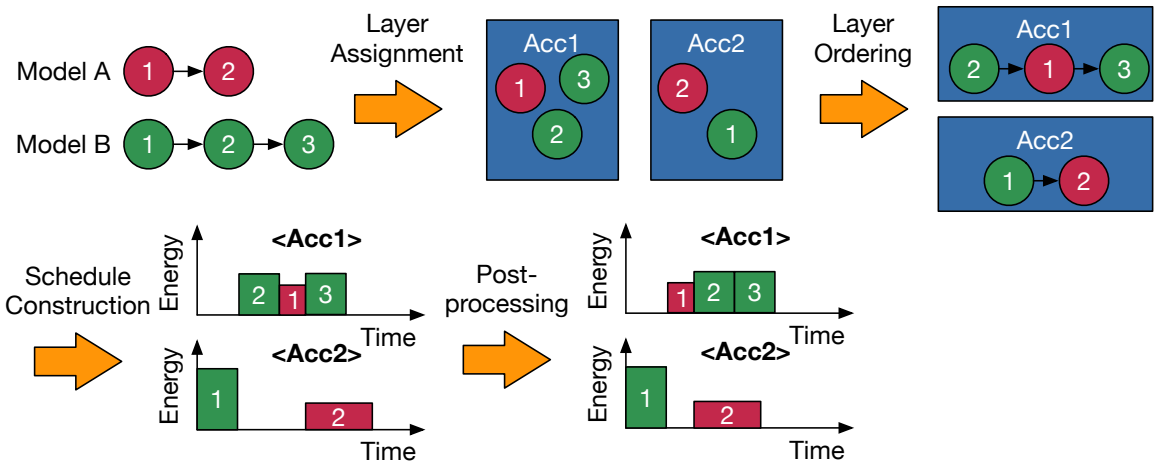


Figure 6.5: An overview of three processes to schedule layers on HDAs, and the boundary of two-phase scheduler implementation of Herald.

6.2.4 Layer Scheduling

Circled numbers represent layers in each model.

The goal of scheduling in Herald is to minimize the energy consumption and latency of an HDA, exploiting different preferences of each layer to accelerators. However, the layer scheduling space is massive. For example, 4.1×10^{152} possible layer execution schedules exist for AR/VR workload A in Table 6.2 across three sub-accelerators in an HDA. To deal with such a large search space, we develop a heuristic the characteristics of DNN workloads to reduce the scheduling overhead. Two major characteristics we exploit are the dependence among layers; layers have mostly linear dependence chain in most models, and they are independent across models. In Figure 6.5, we present an overview of the layer scheduling processes that consist of three processs: layer assignment to each sub-accelerator, layer ordering within each sub-accelerator, and re-ordering as post-procesing.

Herald implements two-step scheduler to perform the three steps

Step 1: Layer Assignment and Ordering. We describe the layer assignment and ordering algorithm in Figure 6.6. The algorithm iterates the frontier layers, (i.e., layers to be executed first based on the dependnce from each model in the workload, until it schedules all the layers. For each frontier layer, the algorithm first queries the cost of execution on each sub accelerator and identify the best-fit sub-accelerator. Figure 6.6 describes energy-delay-

```
Inputs
 - A list of hardware parameters of sub accelerators (Accs)
 - A list of DNN models to run, sorted in the dependence order (MD)
 - Load-balancing factor (LbF)
Outputs
- A list of (schedule time, layer ID, model), (Schedule)
- A list of completion time  for each sub-accelerator (Tot_Latency_Acc)

cycle = 0;
while MD.notEmpty do
  for model in MD do
    layer = model.head;
    // Get EDP/Latency for the layer on each acc
    (EDP, Latency) = MAESTRO_Herald.query(layer, Schedule, Accs) ;
    best_fit_acc = getAccIndex(min(EDP));
    // Check dependence, memory size, and load-balancing conditions
    dependence_cond = is_prev_layer_complelete(Schedule, model, cycle);
    mem_size_cond = MemorySize(cycle, Schedule) + cost.getMemSizeReq < MemorySize;
    load_balance_cond = max(Tot_Latency_Acc) < LbF * (Tot_Latency_Acc[acc] + Latency[best_fit_acc];

    if dependence_cond and mem_size_cond then
      if load_balance_cond then
        ToT_Latency_Acc[best_fit_acc] += Latency[best_fit_acc]; // Assign layer
        PopLayer(MD, layer);
        assigned_a_layer = true;
      else
        //Try the second, third, ... -best fit accelerator for load-balancing
      end if
    end if
    if assigned_a_layer then
      rearrange(MD); // Rearrange the order of model based on the layer ordering
                     //  strategy (depth-first, breadth-first. etc) selected by users
      break;
    end if
  end
  cycle = nextLayerCompletionTime(Schedule) // Failed to schedule; defer execution
end
```

Figure 6.6: Layer assignment and ordering algorithm.

product (EDP) as the metric for example purpose while Herald supports various metrics (e.g., latency, energy, and custom metric from users).

Once identified the best-fit sub-accelerator, the scheduler checks (1) dependence, (2) memory, and (3) load-balancing conditions. The dependence check tests if the execution of previous layer of the layer to schedule is complete or not. The memory check tests if scheduling the layer requires requires memory space more than available at the time to schedule, considering previously scheduled layers. The load-balancing check tests if scheduling the layer results in extreme load balance defined by a user parameter, load-balancing factor, which defines the maximum ratio of the latency of the fastest and slowest

completing sub-accelerators. For example, if the load-balancing factor is two, the slowest sub-accelerator must complete all the scheduled layers within $2 \times Latency_{fastest\_acc}$, where $Latency_{fastest\_acc}$ is the latency of the sub-accelerator completing earilest.

In case only load-balancing check fails, the scheduler try the second-best-fit sub-accelerator until it finds an alternative that meets all the conditions. If not found, the scheduler increment the scheduling cycle to the completion time of a layer on any of sub-accelerators tracked by *Tot_Latency_Acc* in Figure 6.6, which represent the completion time of each sub-accelerator. If all the conditions are met, the scheduler assign the layer onto the best-fit sub-accelerator and remove the scheduled layer from the corresponding model in the model list (MD in Figure 6.6). And then, the reference cycle is incremented in the same way as scheduling fail updated it, and search for another schedulable layer.

Before move on to the next layer, the scheduler change the model to be scheduled next to implement layer-ordering strategy specified by users. Herald supports depth-first and breadth-first ordering. Depth-first ordering schedules all the layers within a model and moves on to the next model. Breadth-first ordering schedules one layer from each model, and repeat it until the scheduler finishes. Such ordering is possible without violating dependences since DNN models mostly have linear dependence for layers. Even if there exists branches like Inception [3], the layers within each branch is also in linear dependence. However, those two ordering methods are mainly for reducing the complexity of scheduling, which do not guarantee the optimality of the resulting schedule. Therefore, after we construct an initial schedule, we perform post-processing to fix inefficiency caused by the simple layer ordering methods.

Step 2: Post-processing. Once an initial schedule is generated by the first step, the scheduler in Herald further optimizes the schedule by exploring layer execution reordering within each sub-accelerator, which fills redundant idle time (i.e., gaps) caused by layer order strategies chosen for Step 1. We describe an example in Figure 6.7. Figure 6.7 (a) and (b) show example initial schedules generated by Step 1 applying depth- and breadth-first layer

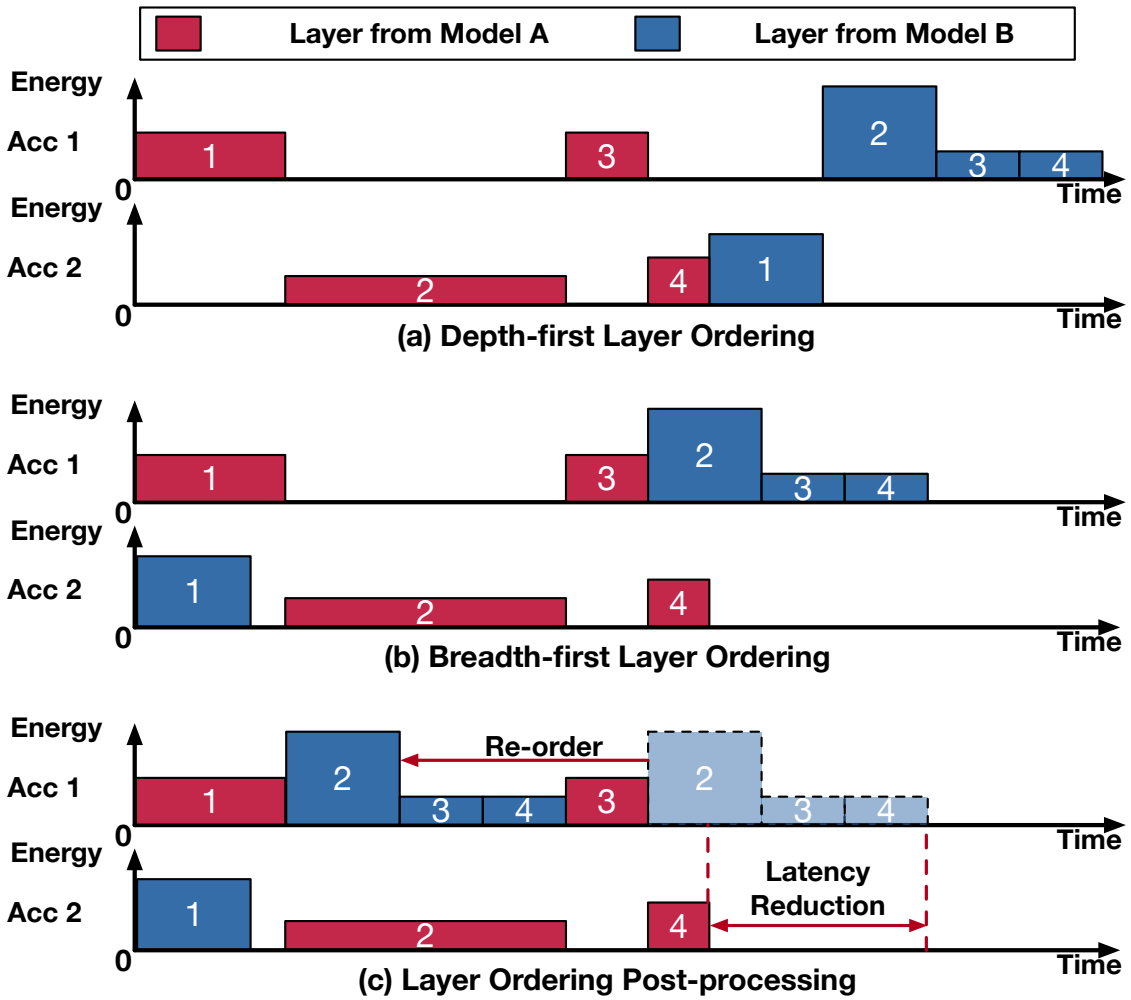


Figure 6.7: Example timelines from different layer ordering methods on two accelerators and two DNN models. Numbers in each box represent the layer number.

ordering, which we discussed in Step 1. As we can observe in the examples, idle time exist in both. However, some of the such idle time is redundant. For example, in Figure 6.7 (b), layer 2,3, and 4 from model B can be scheduled earlier, if that does not violate memory condition, as shown in Figure 6.7 (c).

We describe the post-processing algorithm in Figure 6.8, which eliminates redundant idle time. Post-processing algorithm performs look-ahead of scheduled layers from the completion time of each layer, searching for other schedulable layers within idle time. The algorithm performs similar checks as the schedule did in Step 1 (dependence, memory, and load-balancing). In addition to the three checks, the algorithm also checks if the idle time (i.e., gap of the initial schedule) is sufficient for scheduling fetching a layer from the later in

```
Inputs
 - A list of hardware parameters of sub accelerators (Accs)
 - A list of (schedule time, layer ID, model) (Schedule)
 - Look-ahead depth (LA)
Outputs
 - An updated schedule (Schedule)

for acc in ACCs do
  for baseLayerIdx in NumLayers(Schedule[acc]) do
    look-ahead =1
    while look-ahead  < LA do
      prev_completion_time = Schedule[acc][baseLayerIdx].completion_time
      test_layer = Schedule[acc][baseLayerIdx + look-ahead]
      // Test dependence, memory, load-balancing, and schedule overlap
      if layers is_schedulable(test_layer,  prev_completion_time, Schedule) then
        // Reorder the test layer
        UpdateSchedule(Schedule, test_layer, prev_completion_time)
      end if
    end
  end
end
```

Figure 6.8: Post-processing algorithm that minimizes the idle time.

the schedule to the beginning of the idle time.

6.3 Case Studies

To show the potential of HDAs as future proof, we perform case studies on HDA designs and layer execution schedules generated by Herald using two heterogeneous workloads listed in Table 6.2.

6.3.1 Case Study Settings

Workloads. Based on AR/VR-motivated DNN models listed in Table 6.1, we model AR/VR workloads as listed in Table 6.2. For each DNN model, we assign different number of batches to model different target processing rate of each sub-task. In addition to AR/VR workloads, we also evaluate multi-stream ML-perf inference workload related to computer vision, considering the motivation toward AR/VR.

Mapping. Although Herald can handle arbitrary number of mapping styles in sub-accelerators, we combine two and three distinct mapping styles from recent DNN accelerators(Shi-diannao [61], NVDLA [49]), and Eyeriss [16]. The selection of mapping style is based on

Table 6.2: Heterogeneous DNN workloads used for the evaluation. We model AR/VR workloads using models listed in Table 6.1. We also evaluate computer-vison networks in MLPerf. Number of batches models different target frame rates for each sub-task.

Workload	Model	# of batches
AR/VR-A	Resnet50	2
	UNet	4
	MobileNetV2	4
AR/VR-B	Resnet50	2
	Unet	2
	MobileNetV2	4
	BR-Q Handpose	2
	Focal Length DepthNet	2
MLPerf-CV	Resnet50	1
	MobileNetV1	1
	SSD-Resnet34	1
	SSD-MobileNetV1	1

Table 6.3: Two hardware parameter settings we use for the evaluation. For heterogeneous accelerators, each setting indicate the total amount of hardware resources to be partitioned into sub-accelerators.

Acc. ID	Num. of PEs	NoC BW	Glob. Memory
Small (Edge)	1024	16 GB/s	4 MiB
Medium (Mobile)	4096	64 GB/s	8 MiB
Large (Cloud)	16384	256 GB/s	16 MiB

their distinct parallel unrolling (or, partitioning) strategies to maximize synergy among mapping styles. For example, Shi-diannao parallelizes output row and column over PEs while NVDLA parallelizes input and output channels, which has significantly different preference to layer shapes. Also, Shi-diannao style can run depth-wise convolution more efficiently than NVDLA; NVDLA is optimized for channel-wise accumulation but depth-wise convolution does not have channel-wise accumulation. However, NVDLA provides higher efficiency for fully-connected (FC) layers because FC layers are equivalent to CONV2D layers with only one output activation per each input and output channel. That is, combining those two mapping styles in an HDA and running each layer on an appropriate sub-accelerator can provide latency and energy benefits.

Cost estimation. As we discussed in Section 6.2.2, we extend MAESTRO for the latency and energy estimation.

Accelerators. Based on previously proposed cloud and mobile accelerators, Cloud TPU [17] and Qualcomm Hexagon [116], we select three accelerator settings with the number of

Table 6.4: Hardware resource partitioning optimization results for two-way HDA based on NVDLA and Shi-diannao style accelerators. (NVDLA/Shi-diannao)

Setting	BW partitioning	PE Partitioning
AR/VR-A, Small	4 / 12	128 / 896
AR/VR-A, Medium	40 / 24	1792 / 2304
AR/VR-A, Large	224 / 32	9728 / 6656
AR/VR-B, Small	4 / 12	128 / 896
AR-VR-B, Medium	48 / 16	1536 / 2560
AR/VR-B, Large	128 / 128	12032 / 4352
MLPerf-CV, Small	4 / 12	64 / 960
MLPerf-CV, Medium	32 / 32	1280 / 2816
MLPerf-CV, Large	160 / 96	8192 / 8192

PEs 1K, 4K, and 16K, as described in Table 6.3. We also correspondingly scaled network-on-chip (NoC) bandwidth and global memory. We estimate the extra energy costs for reconfigurability of MAERI based on the open-sourced RTL by running CAD tool chain using a 28nm library like MAESTRO's reference hardware cost model did. We also model homogeneous multi-sub-accelerator chips [117] with evenly partitioned hardware resources and Herald's scheduler.

Schedulers. We apply the scheduling algorithm we discussed in Section 6.2.4 in Herald. We compare the EDP of heterogeneous accelerator designs with the best EDP for each experiment based on a baseline scheduler and Herald's scheduler. The baseline scheduler performs EDP-greedy layer selection and depth-first layer ordering discussed in Section 6.2.4.

6.3.2 Results

Based on the observation from data we collected, we highlight some of them that provide useful insights.

Costs and Benefits of HDA

We estimated latency and energy of HDAs in Figure 6.10, Figure 6.11, and Figure 6.12. We observe that well-optimized HDA points are always on the Pareto curve, and monolithic designs are not. The flexible accelerator was on the Pareto curve on AR/VR-A (small and medium accelerators) and MLPerf-CV (medium accelerator). On average, compared to the

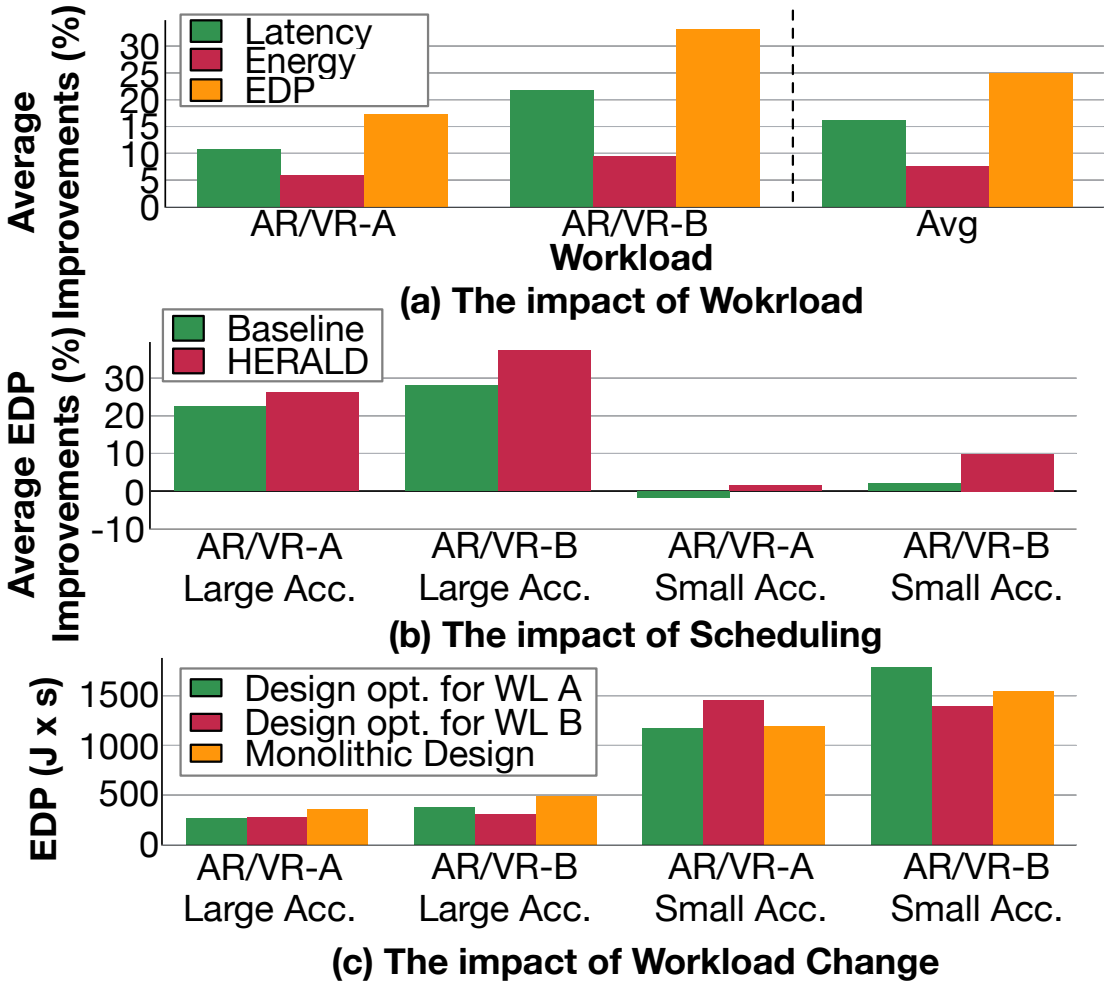


Figure 6.9: The impact of workload and scheduling on the EDP of large and medium HDAs. (a) The average latency, energy, and EDP improvements of HDAs compared to the best monolithic accelerator for each workload. (b) The average EDP improvements compared to the best monolithic design using baseline and Herald's scheduler.

best monolithic design with the lowest EDP, the best heterogeneous design provided 56.0% EDP improvements across all the case studies in Figure 6.10, Figure 6.11, and Figure 6.12.

Optimal HW Resource Partitioning. As we observed in Figure 6.4, naive (or even) partitioning of hardware resources often lead to sub-optimal HDAs. The case study results of optimal hardware partitioning shows the same observation. In Table 6.4, we list the hardware resource partitioning results of design points with the best EDP for two-way HDAs based on NVDLA and Shi-diannao style mappings. We can observe that the optimal hardware partitioning is not trivial, which necessitates a systematic approach like Herald. This is because more number of active PEs requires more bandwidth, and the number of

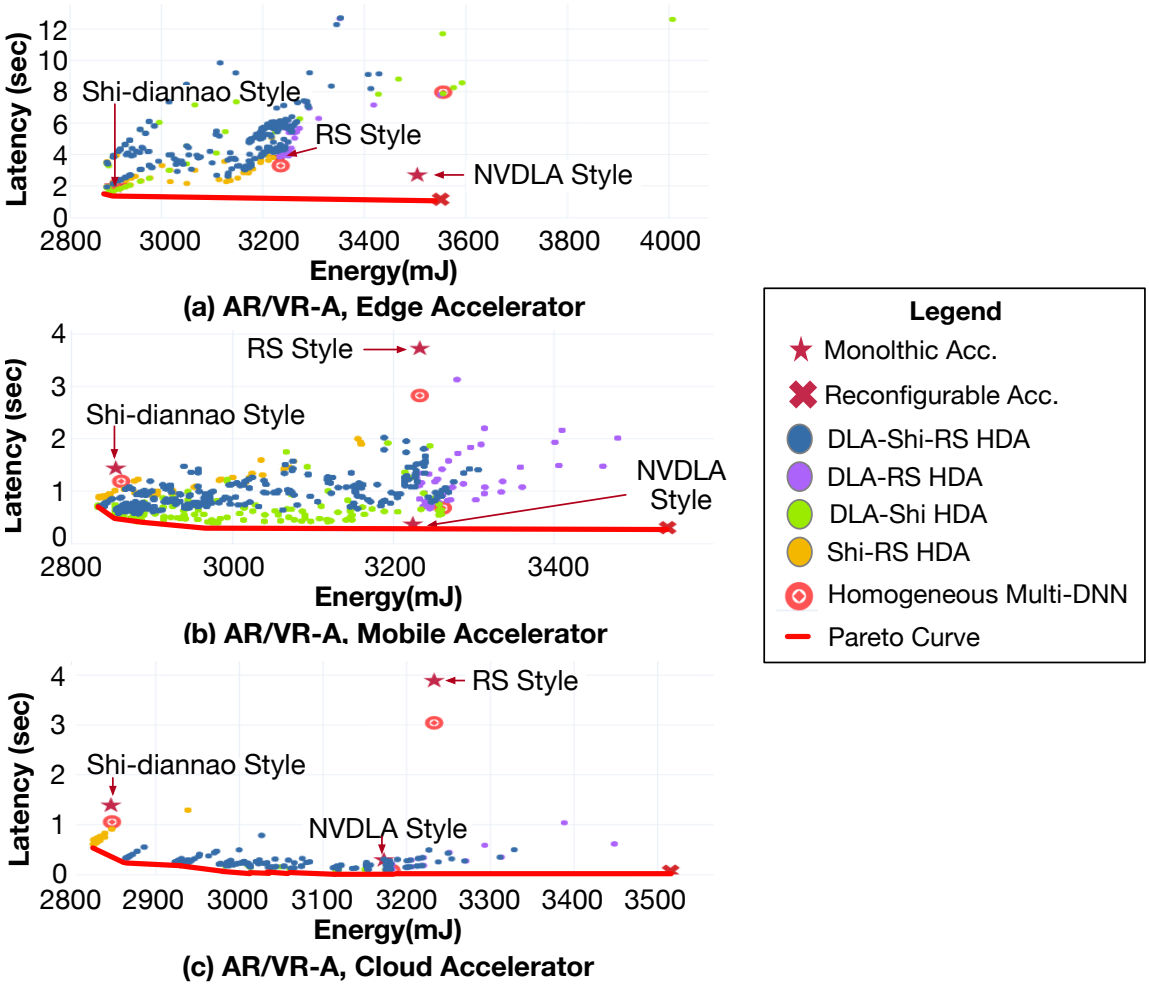


Figure 6.10: Design space of two- and three-way heterogeneous DNN accelerators on the AR/VR-A workload. Each point represents a design point (i.e., HW partitioning choices) with an optimized schedule for the design point. We label each monolithic design point in each plot.

active PEs is a complex high-dimensional function of layer operation, layer size, number of PEs, mapping, and so on [19]. Also, we observe homogeneous multi-DNN design points often provide similar latency and energy as shown in Figure 6.10, Figure 6.11, and Figure 6.12. In such cases, we found that the optimization framework resulted in assigning minimum hardware resources to sub-accelerators except one, and tried to maximize gain from the only large sub-accelerator.

Impact of workloads. Each row in Figure 6.10, Figure 6.11, and Figure 6.12 shows the latency-energy space of monolithic designs, two- and three-way HDAs, and flexible accelerators we evaluate. As expected, the design space depended on the workloads. In particular, we observe that workload with more heterogeneity and layers like AR/VR-B workload is

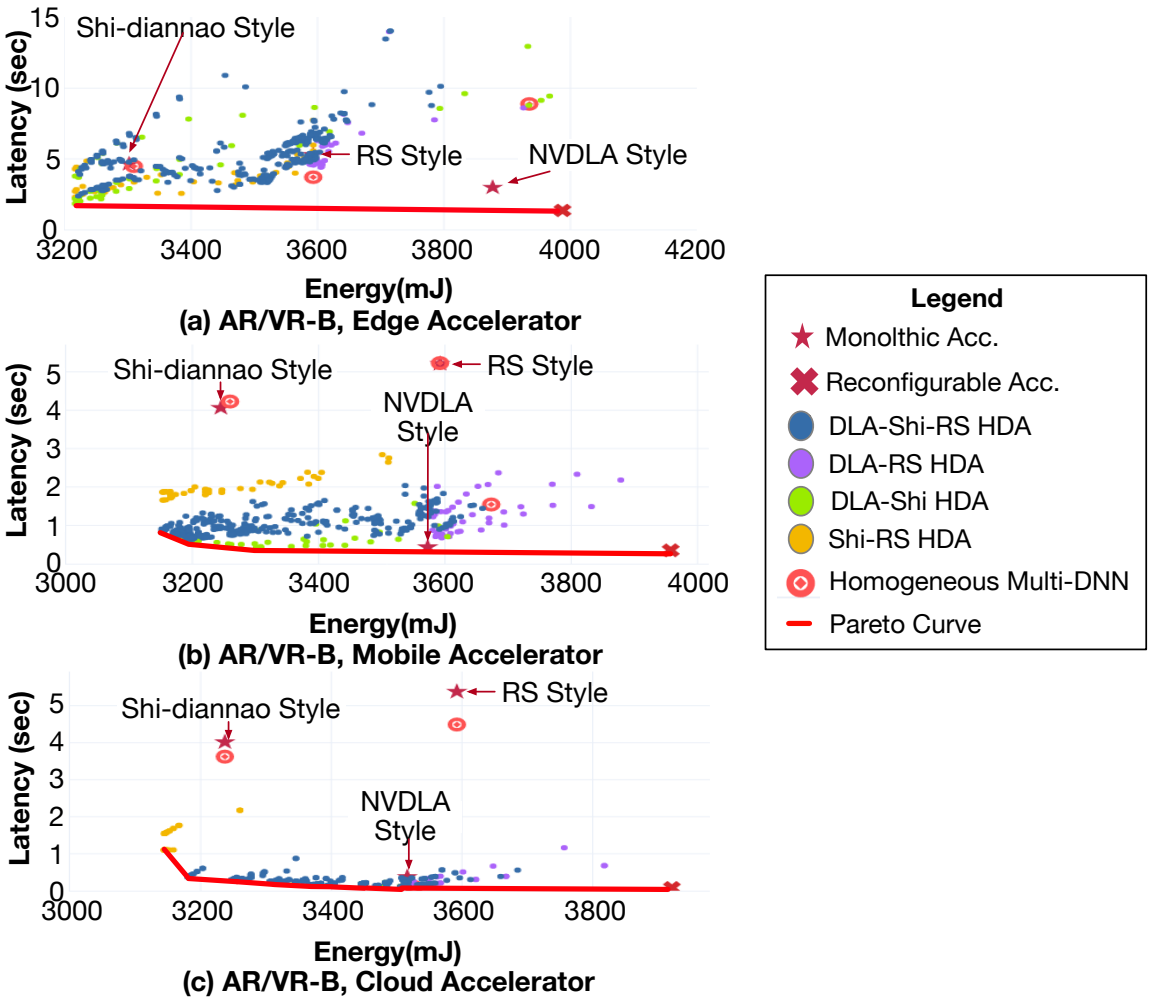


Figure 6.11: The same type of plots as Figure 6.10 for the AR/VR-B workload.

more friendly to HDAs, providing 86.8% latency and 6.61% energy improvements over best monolithic accelerators for each case study in Figure 6.10, Figure 6.11, and Figure 6.12, compared to 63.26% latency and 4.05% energy improvements for AR/VR-A and 3.75% latency and 8.13% energy improvements for MLPerf-CV.

Single-DNN Case. Even for a single DNN, HDAs can still exploit layer parallelism and heterogeneity within a model by batch-processing the workload. We run UNet and Resnet50 using the batch size of four on the large accelerator setting, and plot results in Figure 6.13. We observe that the best monolithic accelerator design is on the Pareto curve, unlike compound workloads we target. However, HDA designs still provide latency and energy benefits over monolithic designs. For UNet and Resnet50 workload, the best HDA provided 26.4% and 48.1% EDP improvements over the best monolithic design. Compared to the

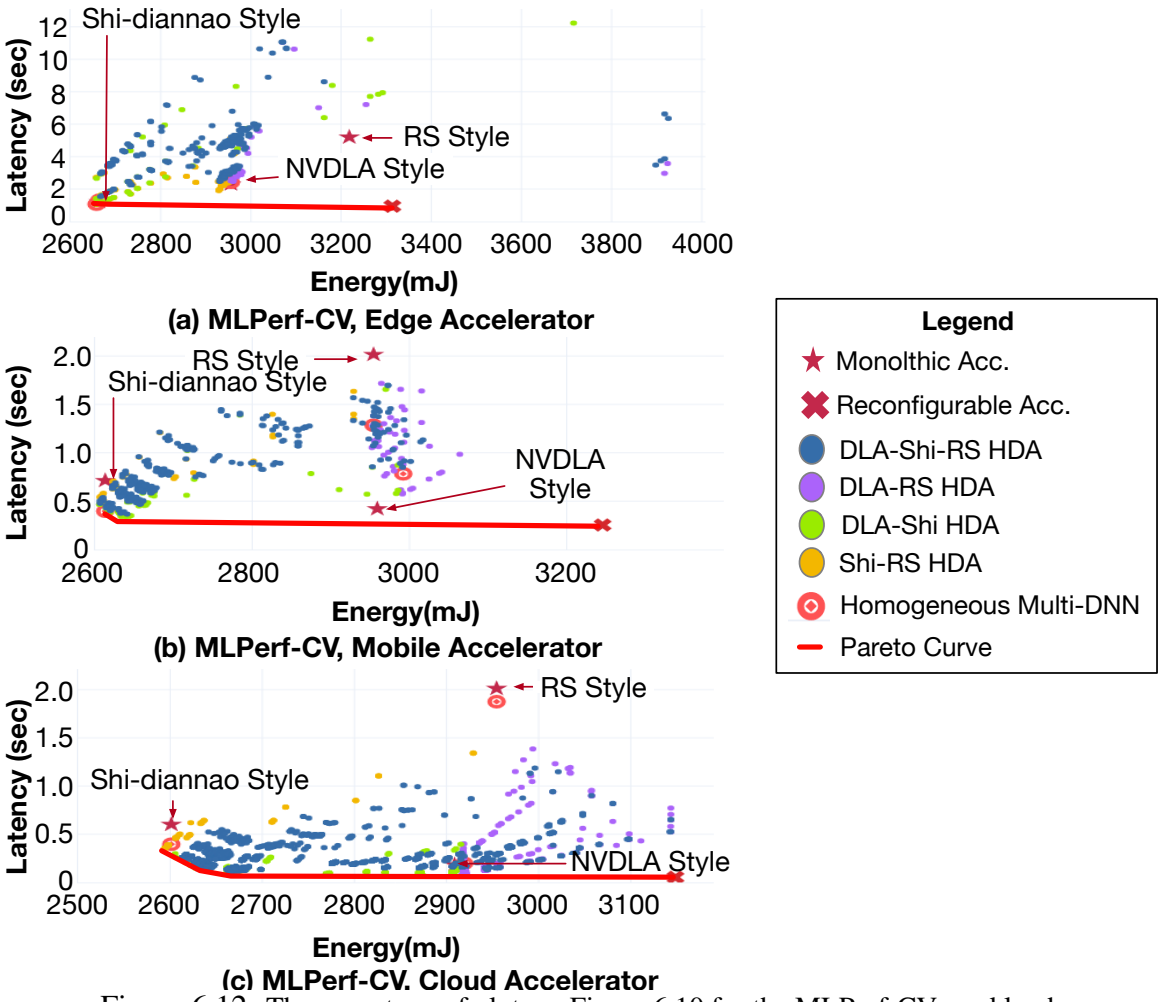


Figure 6.12: The same type of plots as Figure 6.10 for the MLPerf-CV workload.

flexible accelerators, HDA provided 11.7% and 15.8% lower energy for UNet and Resnet50, respectively. For latency, flexible accelerators provided 22.5% and 29.0% lower latency compared to HDAs for UNet and Resnet50, respectively.

Comparison against flexible accelerators. We evaluate a MAERI [22] style flexible accelerator using the same hardware parameter of large accelerator setting. We plot the design point of flexible accelerator in Figure 6.10, Figure 6.11, and Figure 6.12. Compared to the best HDA design points for each evaluation, flexible accelerator designs provided 22.9%, 21.5%, and 24.0% less latency for AR/VR-A, AR/VR-B, and MLPerCV workloads, respectively. However, flexible accelerator designs required 18.7%, 15.5%, and 18.9% more energy for each workload, respectively. The extra energy cost of the flexible accelerator is based on extra hardware components for reconfigurability. In contrast, an HDA can keep

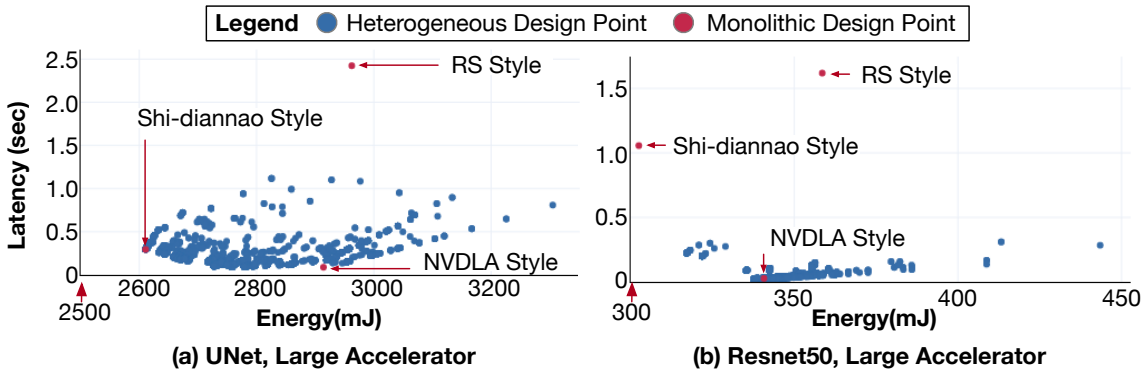


Figure 6.13: Design space on single DNN use cases on (a) UNet and (b) Resnet50 based on large accelerator settings in Table 6.3.

sub-accelerators with relatively simple architecture compared to flexible accelerators, which leads to energy savings we present.

Results in Figure 6.10, Figure 6.11, and Figure 6.12 show that both heterogeneous and flexible accelerators are both Pareto-optimal. HDAs are beneficial for energy, while flexible accelerators are beneficial for latency. The amount of benefits for latency and energy vary depending on the workload. Therefore, the choice of flexible or heterogeneous accelerators depend on the performance goal, energy constraints, and the target workload.

Impact of workload change. Since DNN models evolves and applications change their inner implementation accordingly, workload change can occur after the deployment of an HDA. Figure 6.9 (c) shows a case study of such scenario. That is, the case study represents the use case as a compile time optimizer (i.e., scheduler). We observe that the large accelerator is less sensitive to the workload change still showing benefits against the best monolithic design with 14.1% EDP increase compared to HDAs originally optimized for new workloads on average. However, when the amount of overall hardware resources is smaller (e.g., medium accelerator), 26.2% of EDP increase compared to HDAs originally optimized for new workloads on average. The results imply that heterogeneous DNN accelerators prefer large accelerators.

Execution time of Herald. Although the scheduling in Herald is designed to be offline, the scheduler is light-weighted. We run Herald on a laptop with i9-9880H processor and 16GB of memory and present the time required for scheduling on each hardware design

Table 6.5: Average time required for scheduling for each hardware design point (i.e., HW partition choice).

Workload	Layers	Number of Sub-accelerators	Time per HW DP (s)
AR/VR-A	448	2	2.89
AR/VR-A	448	3	4.32
AR/VR-B	618	2	3.98
AR/VR-B	618	3	10.74
MLPerf-CV	168	2	1.17
MLPerf-CV	168	3	1.67

point in Table 6.5, since overall runtime heavily depends on user parameters (e.g., search granularity, strategy, etc.). On average, the scheduling requires 9.48 ms per layer and per HDA design point.

Summary. We summarize our main observations below:

- The design space of HDA is not trivial, which requires a systematic co-optimization of hardware resource partitioning and layer execution schedule.
- HDAs outperform or match the monolithic and flexible accelerators, as Pareto curves in Figure 6.10, Figure 6.11, and Figure 6.12 show.
- HDA and flexible accelerators are beneficial for energy efficiency and latency, respectively, while maintaining similar overall EDP (often both of them are on the Pareto curve in latency-energy space).
- Simple combination of homogeneous sub-accelerators does not provide Pareto-optimal design points.

6.4 Summary

In this chapter, we explored the latency and energy optimization opportunities of heterogeneous DNN accelerators on realtime and heterogeneous workloads. Because the efficiency of a DNN accelerator depends on mapping, workload, and hardware design parameters (number of PEs, memory size, memory/NoC bandwidth, etc.), identifying the best heterogeneous DNN accelerator design point with an optimized schedule is challenging. Therefore, we developed Herald, an automated design space exploration and layer scheduler framework

for heterogeneous DNN accelerators. In our case studies, Herald identified optimized design points and layer schedules, providing 56.0% EDP benefits compared to the best monolithic design we compare. Herald has presented that the most efficient design point has non-trivial hardware resource partitioning and a naive scheduler can result in EDP degradation, motivating the necessity of Herald.

Until this chapter, we discussed the potential benefits of supporting flexible mapping, and two approaches to enable flexible mapping on accelerators: reconfigurability and heterogeneity. In the next chapter, we summarize this thesis and discuss future research directions from this thesis.

CHAPTER 7
CONCLUSION AND FUTURE WORKS

7.1 Summary of Contributions

Like deep neural network was enabled by advancement in computer hardware after decades it was first invented, computational performance and efficiency improvements for DNNs can enable new technologies and use cases (e.g., software engineering can be automated via AI [118]). Therefore, DNN acceleration is currently one of the most active research topics in both industry and academia. However, since DNN model changes rapidly, adapting DNN acceleration stack to such changes is challenging.

To deal with such a challenge, this thesis observes that flexibility in hardware to support various DNN models and their layers via various mapping (or, dataflow) styles is the key. In that context, this thesis makes the following contributions:

Data-centric description of dataflows and a light-weight cost model based on the description: Although previous works used loop nests to describe various dataflow styles, loop nests require complicated analysis steps to infer data movements because loop nests describe computation and treat data movement as an implicit aspect. Since data movement dominates in the energy cost of DNN accelerators, loop nest-based analysis frameworks often result in heavy frameworks, which hinders it to be used as a cost model for various optimization frameworks (e.g., dataflow search, hardware DSE, neural architecture search, etc.). Thus, this thesis proposed to directly describe the data movements using three data-centric directives and presented the capability of data-centric directives by showing description of various state-of-the-art dataflows. Also, this thesis presented MAESTRO, a microarchitectural cost model based on the data-centric dataflow descriptions, which made MAESTRO a light-weight cost model with high accuracy. Using MAESTRO, this thesis

presented a hardware design space exploration framework, which performs fast design space search based on the light-weight cost model, MAESTRO.

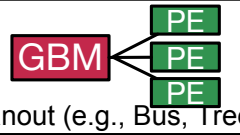
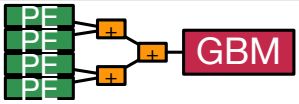
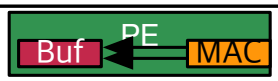
Light-weight NoC specialized for DNN accelerators: This thesis observed that providing flexible data movement is the key for flexibility of DNN accelerators. This thesis presented that traditional NoC solutions used in CMPs do not scale in either NoC performance or hardware costs. Therefore, this thesis proposed a light-weight NoC, Microswitch NoC, specialized for DNN accelerator traffic. This thesis is the first work that explored new NoC topology other than traditional NoCs (e.g., mesh or bus) and NoC switch architecture in DNN accelerators.

Reconfigurable in-network-processing structure for efficient computation of irregular DNNs: Irregular neuron sizes from diverse DNN layer sizes and algorithmic optimizations (e.g., cross-layer mapping, sparsity, etc.) emerged as a new challenge to DNN accelerators since one of the major strategies of specialization in accelerators is proper sizing of hardware for target application. Between two operations, multiplication and addition (reduction), the source of the challenge is in reduction operation because the degree of reduction operation changes based on the neuron sizes while multiplication can be individually computed without any dependence.

To deal with challenges from reduction operations in irregular DNNs, this thesis proposed a reconfigurable in-network-processing reduction tree named augmented reduction tree (ART). ART enabled a DNN accelerator that can map any irregular sized neurons without underutilization due to the irregularity, which results in high compuitational performance for irregular neurons (e.g., sparsity).

Heterogeneous DNN Accelerator Optimization Framework: From another angle for flexibility other than reconfigurability, this thesis explored heterogeneity in a DNN accelerator. A heterogeneous accelerator contains multiple sub-accelerators with different amount of hardware resources that runs different dataflows. By running each layer on the most efficient sub-accelerator, heterogeneous accelerators achieve high efficiency. However, the hardware

Table 7.1: The impact of mapping choices, summarized as reuse type each mapping exploits, on hardware requirements. Note - by temporal multicast, we refer to stationary buffers from which the same data is read over time

Reuse Type	Communication Type	HW Implementation Choices	
Spatial	Multicast	 Fanout (e.g., Bus, Tree)	 Store-and-Fwd (e.g., Systolic Array)
	Reduction	 Fanin (e.g., Reduction Tree)	 Reduce-and-Fwd (e.g., Systolic Array)
Temporal	Multicast	 Multiple reads from a buffer	
	Reduction	 Multiple read-modify-write to a buffer	

resource partitioning across sub-accelerators and scheduling layers on sub-accelerators determine the overall efficiency of a heterogeneous accelerator, and it is not a trivial task because of the complexity of the problem (e.g., scheduling is an NP-hard problem).

This thesis proposes solutions to the challenges by reducing the complexity using domain knowledge of DNNs and using light-weight cost model, MAESTRO. This thesis presents a heterogeneous accelerator optimization framework that ensures users to extract maximum potential gains from the heterogeneous accelerator system.

7.2 Implication of Hardware Choices on Mapping

In Section 3.2.3, we discussed what types of hardware supports are required for various mappings. Such analysis is based on a design flow for flexible accelerator: (1) **[Mapping to hardware flow]** we first find optimized mappings for target layers, and design an accelerator that can support the mappings (i.e., find the algorithmic optimum first, and try to achieve it). Table 7.1 shows the reuse types from various mappings and supporting hardware.

However, an alternative design flow is also possible: (2) **[Hardware to mapping flow]**

Table 7.2: The impact of buffer choices on mapping. We omit the constraint based on buffer size because it is not hardware choice-specific.

Hardware Choice	Latency	Hardware Cost	Implication to Mapping							
			Mapping Size						Spatial Dimension	Iteration Order
			K	C	Y	X	R	S		
FIFO	Rd/Wr: **O(1)**	Control: O(1) RAM: O(BufSize)	-	-	≤Sz(R)	≤Sz(S)	-	-	-	-
Scratchpad	Rd/Wr: **O(1)**	Control: O(n) RAM: O(BufSize)	-	-	-	-	-	-	-	-

we first design an accelerator and find an optimized mapping on the accelerator (i.e., design hardware within available budget and maximize the efficiency of the hardware).

For the second flow, the hardware to mapping flow, we need to understand how hardware components constrain possible mappings, which is the opposite way we analyzed their relationship in Section 3.2.3. That is, the hardware components collectively decides the degree of mapping flexibility of the resulting accelerator. At the same time, we also need to understand the costs and benefits of the flexibility enabled by hardware component choices. Therefore, we discuss the implication of hardware components on mappings, performance, and hardware costs.

Buffer. We discuss two possible hardware choices for buffers: FIFO (First-In-First-Out) and scratchpad.

FIFO is popular for low-level memories (near PEs) due to its simple control circuit. However, the major restriction on FIFO is the lack of addressing; always reads the top data and writes the tail data. Unless we allow circulation (which we assume in this analysis), FIFO cannot access past data point once a new data point is read from the FIFO. In CONV2D operation, such a limitation imposes constraints to available mapping sizes as presented in red texts in Table 7.2. The constraint states that FIFO cannot hold input activation data points across multiple sliding windows (receptive fields) since the same set of filter value needs to be accessed for each sliding window (i.e., requires to access previously accessed data).

Scratchpad is the most popular choice for buffers in DNN accelerators since it provides full addressing (i.e., read/write from/to any address) and full control of actions to program-

Table 7.3: The impact of distribution NoC choices on mapping.

Hardware Choice	Latency	Hardware Cost	Implication to Mapping							
			Mapping Size						Spatial Dimension	Iteration Order
			K	C	Y	X	R	S		
Bus	**O(** D/m / BW **)**	O(n)	-	-	-	-	-	-	C Unpreferred	SpDim: inner-most position unpreferred
Crossbar	**O(** D / m / BW **)**	$O(n^2)$	-	-	-	-	-	-	-	-
Store-and-Forward	**O(** max(D/m, n) / BW **)**	O(n)	-	-	-	-	-	-	-	-
Tree	**O(** max(D/m, log(n)) / BW **)**	O(n log(n))	-	-	-	-	-	-	C Unpreferred	SpDim: inner-most position unpreferred
Mesh (M-cast)	**O(** max(D/m , √n) / BW **)**	O(n)	-	-	-	-	-	-	-	-

- n : Number of PEs (sub-clusters)
- m: Multicast Factor
- D: Total number of data points to distribute
- BW: Distribution NoC bandwidth

mers, which enables programmers to exploit the knowledge of the target application. Based on the full addressing capability, scratchpad does not impose any constraint on mapping (except one from buffer size).

Note that cache memory is rarely employed in accelerators since (1) staging decision is implicit, which does not allow fine-grained optimization based on the knowledge of the target application [119], and (2) the hardware cost for implicit staging decision is high (e.g., LRU eviction algorithm).

Finally, for all buffers, mapping size is constrained by buffer sizes. We describe the constraint from buffer size in Theorem 1, in the appendix.

Distribution NoC. The core capability in distribution NoC is multicast capability. However, its impact is more on performance and energy, not on constraint toward the mapping. Such an aspect appears as preference toward specific types of mappings. For example, bus and tree natively supports multicast. On such NoCs, mappings that exploit the multicast capability are preferred, as described in spatial dimesnion and iteration order columns of Table 7.3. For example, if input channel (C) dimension is spatially mapped (or, parallelized), both input activation and filter change across PEs, disabling multicast opportunities. Such a mapping is not preferred by multicast-support NoCs.

For distribution NoCs, understanding the trade-off between latency and hardware cost is

Table 7.4: The impact of reduction NoC choices on mapping.

Hardware Choice	Latency	Hardware Cost	Implication to Mapping							
			Mapping Size						Spatial Dimension	Iteration Order
			K	C	Y	X	R	S		
Temporal Reduction	**O(** R/n * rt / BW **)**	O(1)	-	-	-	-	-	-	C/R/S Prohibited	-
Reduce-and-Forward	**O(** max(R/rs, n) / BW **)**	O(n)	-	-	-	-	-	-	C/R/S Preferred	-
Adder Tree	**O(** max(R/rs , log(n)) / BW **)**	O(n log(n))	-	-	-	-	-	-	C/R/S Preferred	-

- n : Number of PEs (sub-clusters)
- rs/rt : Spatial/Temporal Reduction Factor
- R: Total number of data points (i.e., partial sums) to reduce
- BW: Reduction NoC bandwidth

crucial. We summarize the trade off in latency and hardware cost columns of Table 7.3. The latency in Table 7.3 considers multicasting and pipelining effect in NoCs (e.g., in systolic array-style distribution NoC, or store-and-forward, we can inject new data points every cycle even if previous data did not arrive their destination). For mesh, we assume store-and-clone style multicast support, like store-and-forward NoCs (i.e., systolic array-style).

Reduction NoC. The core capability in reduction NoC is spatial reduction capability. For example, the temporal reduction does not support reduction across PEs, while adder tree or reduce-and-forward (systolic array style) support it. The lack of spatial reduction capability prohibits a mapping that implies reduction across PEs. In contrast, NoCs with spatial reduction capability prefers such a mapping.

The spatial dimension column of Table 7.4 summarizes the constraint and preference toward spatial reduction. In CONV2D operation, input channel (C), filter row (R), and filter column (S) dimensions need to be accumulated to generate an output. Therefore, they cannot be spatially mapped unless the reduction NoC in the target accelerator supports spatial reduction.

Like distribution NoCs, reduction NoCs are also in trade-off space between latency and hardware costs. We summarize such trade-off in latency and hardware cost columns of Table 7.4. Similar to distribution NoCs, we consider the pipelining effect for latency. Note that temporal and spatial reduction factors are separate for reduction NoC analysis, unlike multicast factor for distribution NoC accounts for the spatial part only.

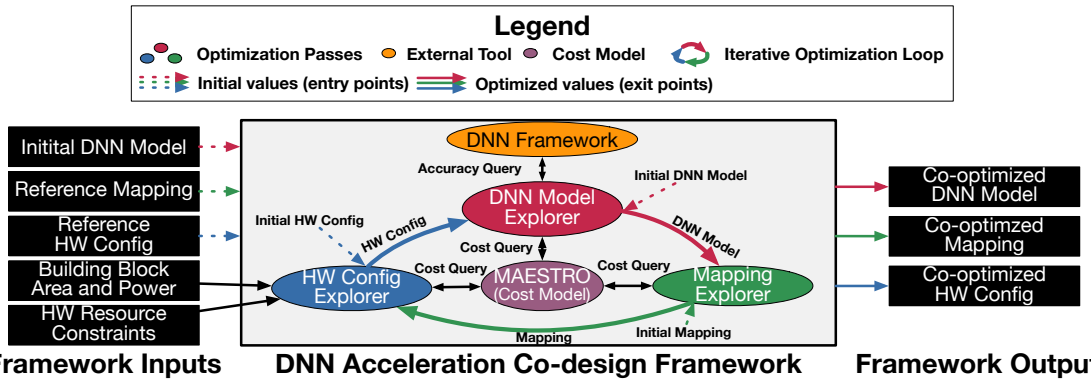


Figure 7.1: MAESTRO as a cost model for a future DNN algorithm - mapping - hardware (i.e., full stack) co-design framework for DNN acceleration.

To fully understand the accelerator design space, exploring two possible design flows (mapping-to-hardware and hardware-to-mapping) is helpful, which provides different strength and weakness. For the hardware-to-mapping approach, we need to understand the impact of hardware choices on mapping flexibility and the costs and benefits of the flexibility. This section presented a high-level analysis of such impacts. We could observe that the trade-off is not trivial, which motivates another systemic approach like MAESTRO Chapter 3 that can guide the hardware-to-mapping design flow. By such an approach, hardware designers will be able to decide the right amount of mapping flexibility of their target workloads, which will minimize hardware costs for unused features for extra flexibility. We discuss such future research directions next.

7.3 Future Directions

Based on this thesis, we envision the following as promising research direction.

7.3.1 Optimizing Flexibility

MAERI provides full flexibility to support any size of neurons. However, such degree of flexibility is not required by all applications. How do we identify the right amount of flexibility in hardware for a target application, as we discussed in Section 7.2? We envision incorporating the flexibility metric we define in Section A.1 to MAESTRO and augmenting MAESTRO with detailed cost model for hardware component choices will provide us a

framework to guide flexible accelerator design.

7.3.2 Data-Orchestration Aware Compilers

The true benefit of any DNN accelerator can only come from an optimized mapping of the DNN over it. Compilers for DNN accelerators use mapping auto-tuners which often rely on simple heuristic/cost models. However, because the mapping space is huge (e.g., Resnet50 has 10^{20} possible mappings on a 256 PE accelerator) so light-weight cost model is crucial. Therefore, MAESTRO as a cost-model can enable efficient mapping strategies for arbitrary DNNs and hardware platforms, and be plugged into existing compilers like XLA and TVM.

7.3.3 Reuse-aware Neural Architecture Search (NAS)

As AI becomes pervasive, we expect DNNs to run on device form factors ranging from tiny sensors to datacenter racks. Naturally, finding the right DNN model for each platform is an open question today, and the goal of NAS. Most NAS frameworks measure the efficiency of models only using the number of multiply-and-accumulate (MAC) operations, which can mislead the optimization direction since actual cost is in a complex trade-off space of DNN model, mapping, and hardware, as this work shows. We envision MAESTRO to be a plug-in in NAS frameworks performing systematic trade-off studies between accuracy and efficiency for arbitrary HW platforms.

7.3.4 Full-stack Co-design for DNN Acceleration

As accelerator designs are tightly coupled with application to accelerator because of their design philosophy, specialization, future research in DNN acceleration need to optimize all of the aspects in the computation stack, from hardware to application levels.

MAESTRO has the potential to enable DNN-mapping-hardware co-design, as shown in Figure 7.1, as a comprehensive cost model. We believe such co-design will be the ultimate goal for DNN acceleration, which will provide us optimality beyond current optimal points,

and MAESTRO can play a key role in co-design frameworks like we show in Figure 7.1.

7.3.5 Cost-Benefit Modeling of Mappings on Accelerators for HPC Applications

Data-centric specification of data movement can help the HPC community. HPC systems deal with large distributed systems where minimizing data movement is crucial. Moreover, the most time consuming HPC kernels (for e.g., climate, physics, quantum chemistry, oil and gas simulations) are based on linear algebra primitives such as matrix multiplications, lower/higher-order stencils, and tensor contractions (higher-dimensional generalizations of matrix-matrix multiplication), which the data-centric representation can support without additional extensions. We envision the data-centric directives augmenting OpenMP pragmas to enable HPC programmers to write efficient code for emerging accelerator-based systems. MAESTRO can also be extended to model the the cost for local vs remote accesses, extending its hardware representations from spatial accelerators to large distributed systems.

7.3.6 Communication-driven Design Methodology

MAERI specialize its architecture for on-chip communication as well as computation for deep learning workload while most of accelerators mainly focus on computation. To distribute both of computation and communication, we integrate computation in interconnects, which let accelerators offload computation in interconnects. This constructs "*bump-in-the-wire design*," or in-network-processing. We believe such communication-centric design methodology can help many accelerators for various applications to reduce both communication and computation delays and costs, which will reveal new horizon of accelerator optimization.

7.3.7 Microswitches for Accelerators Targeting Applications Other than DNNs

The communication-centric approach of Microswitch NoCs can be extended to applications other than DNNs since the data movement cost (in both pefrmance and energy) dominates

in most accelerators. Microswitch-B can be an ideal candidate for other applications because of its composability. Therefore, we envision a NoC generation framework based on Microswitch-B that can provide a optimized NoC topology for any application composed by Microswitch-B.

7.3.8 ART Topology in Different Granularity

The ART topology of MAERI can be used in a finer-grained way to implement flexible bit precision adders by organizing one-bit adders using ART. Also, it can be used in a coarser-grained way to implement data center-scale distributed systems, which will enable efficient spatial and bulk map-reduce or deep learning task processing. In such distributed systems, users can apply software-defined network (SDN) paradigm to program MAERI as SDN concept is innate in MAERI.

7.3.9 General Map-Reduce Accelerators

Although we presented MAERI as a specialized architecture for deep learning applications, the architecture can be used for any application that has *map-reduce* patterns. To support a map-reduce application, we just need to replace compute units in multiplier and adder switch to those of target map and reduce functions, respectively. For example, many graph applications iterate over frontier nodes of a source node and compute application-specific scores of each frontier (map operation). Then we compute the next source node based on scores from frontier nodes (reduce) and iterate to the next source node. Such patterns occur in graph applications such as parallel depth- and breadth-first search, graph clustering, minimum spanning trees, maximum flow, and so on. Future works can explore variances of MAERI architecture specialized further for such graph applications.

7.3.10 Efficient Sparsity Support in DNN Accelerators

The capability of MAERI to efficiently support irregular neurons with various sizes at the same time suggests that MAERI can be extremely efficient architecture for sparse workloads. Although we only explored weight sparsity support using MAERI, with extra structure for weight-input index matching, MAERI architecture can be extended to support input sparsity or both of weight and input sparsity.

Appendices

APPENDIX A
FLEXIS: ESTIMATING THE DEGREE OF FLEXIBILITY IN A RECONFIGURABLE DNN ACCELERATORS

In Chapter 4 and Chapter 5, we explored the reconfigurability approach for enabling flexible mapping on DNN accelerator with a reconfigurable NoC to efficienctly support arbitraery DNN accelerator traffic and a communication-centric reconfigurable DNN accelerator design that provides rich flexibility to run any DNN workloads with irregular neuron sizes. In addition to the works this thesis present, many reconfigurable accelerators have been proposed [16, 21, 27] based on various implementation choices.

Although all of such works presented promising results with reconfigurable accelerators, we currently lack a unified methodology to compare the degree of flexibility each design provides and analyze the costs and benefits of them. Therefore, in this Appendix, we propose a metric of mapping flexibility in DNN accelerators, and clarify how the choice of hardware components constraint possible mappings on DNN accelerators.

A.1 Mapping Flexibility

Since all DNN accelerators compute DNNs, the definition of flexibility based on functionality (i.e., what they can compute) does not provide distinction among flexible accelerators. Instead of supporting more applications, the focus of flexible accelerators is providing capabilities for supporting various mapping styles [21, 22, 120]. Therefore, we also focus on the number of supported mappings to compute a target DNN. Considering that the total number of possible mappings on a unconstrained hardware depends on layer operation and sizes, we define the flexibility in layer granularity.

As discussed in Section 2.4, mapping consists of three components as follows:

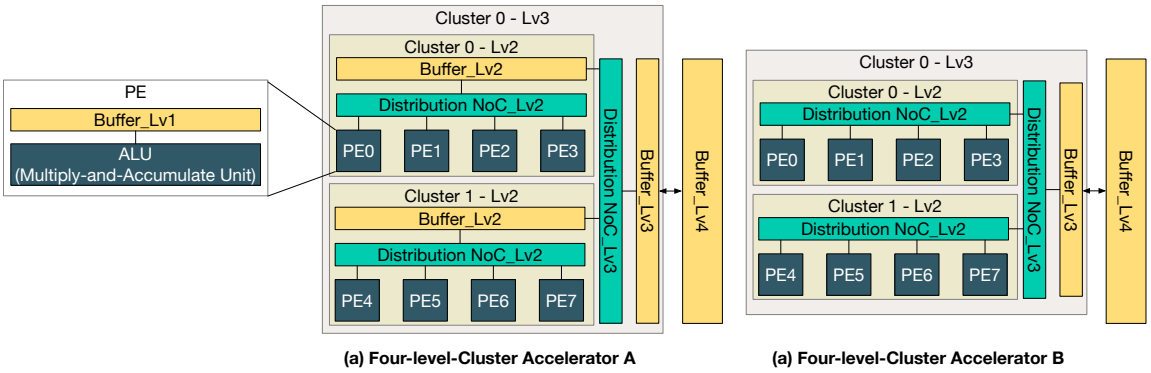


Figure A.1: Cluster levels in two example DNN accelerators. (a) All the clusters are spatio-temporal. (b) Cluster level2 is only spatial (without buffers). Note that Lv0 and Lv4 are purely temporal cluster (buffer-only), which are PE and DRAM.

- Loop order: The order of data dimension iterations. Equivalent to the directive order in data-centric representation.
- Loop parallelization: The data dimension to be parallelized. Equivalent to the choice of spatially mapped dimension in data-centric representation.
- Loop tiling: The number of tiling levels and tile sizing (blocking). Equivalent to the choice of mapping size in data-centric representation.

Based on the three components, we can define (1) loop order flexibility, (2) loop parallelization flexibility, and (3) tiling flexibility. We will define mapping flexibility as the cross-product of all of the three components. We construct formal definition based on data-centric directives we discussed in Chapter 3. Although the definition can be applied to many DNN operations, we use CONV2D operation as the target operation for simplicity. We follow the convention in Figure 2.12 for CONV2D operation.

A.1.1 Base Definitions

We define various information necessary for defining mapping flexibility.

Hardware

As the basic unit of hardware information, we use *cluster*, which refer to a group of hardware components in an accelerator architecture. For example, in Figure A.1 (b), cluster 0 - Lv2

include PE0, PE1, PE2, PE3, and a distribution NoC_Lv2.

Cluster level (Lv) refers to temporal, spatial, or spatio-temporal data distribution/reduction points in a hardware component hierarchy, which includes buffers (temporal) and PE array dimensions (spatial). That is, cluster levels refer to all the temporal (i.e., memory) and spatial (i.e., PE array dimensions) hierarchy levels. Figure A.1 shows cluster levels in two example accelerators. For simplicity, we only present hardware components for the downstream (i.e., DRAM to PE) data orchestration in Figure A.1.

We can compose any hardware component hierarchy using clusters. In Figure A.1 (a), cluster levels 2 and 3 contains a buffer and a distribution NoC that targets four sub-clusters (PE) and two sub-clusters (cluster0-Lv2 and cluster1-Lv2), respectively. Note that PEs are also sub-clusters at level 2 (e.g., PE1 is cluster1-Lv1 in Cluster0-Lv2, PE6 is cluster2-Lv2 in Cluster1-Lv2). Cluster level 0 and 4 are purely temporal, which contain buffers and have only one sub-cluster.

In Figure A.1 (b), cluster level 2 is purely spatial, which does not have a buffer. The abstract architecture in Figure A.1 (b), logically models a 2D PE array like a systolic array (e.g., TPUv1 [17] does not have PE row or column buffer).

Since clusters are recursively organized using the notion of sub-clusters, we can construct arbitrary accelerator architectures with uniform hardware resource distribution across clusters, which holds for many accelerators.

We utilize the concept of cluster levels to model various accelerators. Formally, we define the cluster as follows:

Definition 1. Cluster

A cluster *Clt* in an accelerator is defined as a set of attributes as follows:

$Clt = \{C_{lv}$=INT,

$dist\_temporal$=($Tensor$=Input Tensor, $BufferType$, size=INT) list,

$dist\_spatial$=(Dist NoC Type, size=INT, bandwidth=INT) list,

rdct_temporal=(*Tensor*=Output Tensor, *BufferType*, size=INT) list,

rdct_spatial=(Dist NoC Type, size=INT, bandwidth=INT) list }

In Definition 1, C_{lv} refers to the cluster level, *dist_temporal* and *rdct_temporal* specify the buffer type for distribution and reduction traffic (i.e., for input and output tensors, respectively). *dist_spatial* and *rdct_spatial* specify the NoC type for distribution and reduction, respectively. Overall, cluster defines the buffer-NoC-sub-cluster architecture at each hierarchy level, treating other components in lower levels as sub-clusters (i.e., black boxes). Composing clusters, we can specify an accelerator as follows:

Definition 2. Accelerator

An accelerator *Acc* is an array of clusters (*Clt*):

$Acc = Cltarray$

The indices to array elements refer to cluster levels. For example, Acc[1] refers to PE (level 1 cluster). Acc[4] in Figure A.1 (a) refers to DRAM (level 4 cluster). Note that each entry does not refer to an instance of a cluster (e.g., cluster 1 -Lv2 in Figure A.1 (b)) but specify the architecture at the corresponding level.

Now we define base concepts related to DNN layers and mappings.

Layer and Mapping

Definition 3. Layer Size

Layer size L_{sz} is a set of (dim=*STRING*, size=*INT*) tuples.

Based on the definition in Definition 3, we can define L_{sz} for CONV2D layers as follows:

$$L_{sz} = \{(dim, size) | dim \in \{K, C, Y, X, R, S\} \wedge size \in INT\}$$

Definition 4. Tile Size

The tile sizes of each data dimension at cluster level $C_{lv}C$, $\mathrm{T}(C_{lv})$ is a set of (dim=*STRING*, size=*INT*) tuples.

Note that Definition 4 is similar to Definition 3 since L_{sz} is the tile size at the top-most memory hierarchy that contains entire data points of a layer. Based on that observation, we set the DRAM as the top level, as described in Assumption 1.

Assumption 1. DRAM and Layer Size

Let N_C is the number of cluster levels on an accelerator chip, then $\mathrm{T}(N_C + 1) = L_{sz}$.

Assumption 1 indicates that DRAM houses all the data points (input activation, filter, and output activation) and sets the DRAM as the last level of interest for modeling flexibility. Assumption 1 is based on the size of recent DRAM modules (¿4GB, conservvatively) and CONV2D layer sizes in popular CNN models (e.g., Resnet50 [14]) that typically requires less than 100MB of memory for entire data for a layer.

Definition 5. Full Data Index

For a given layer size L_{sz}, full data index $FIdx$ is defined as follows:

$$FIdx = \{(dim, idx) | \exists (layer_{dim}, size) \in L_{sz},\ s.t.\ dim = layer_{dim} \wedge 0 \leq idx < size\}$$

Full data index is a listing of all the independent data dimensions and their sizes. Recall that some data dimensions can be coupled with multiple tensors (e.g., in CONV2D, input channel(C) is coupled with both input activation and filter). Full data index list up tensor subscripts without duplication. For example, in CONV2D, a full data index (K,0), (C,0), (Y,1), (X,3), (R,0), (S,1) refers to data access toward filter F[k=0][c=0][r=0][s=1], input

activation I[c=0][y=1][x=3], and output activation O[k=0][y'=y-r=1][x'=x-s=2]. That is, full data index specify one data point in each tensor, which are accessed together for a computation.

Definition 6. Overlaid Dimensions

Given a data point of a tensor accessible by a full data index, $FIdx_1$, if the same data point is accessible by simultaneously modifying two data indices $(dim1, idx1), (dim2, idx2) \in FIdx_1$, $dim1$ and $dim2$ are overlaid.

Definition 6 defines overlaid dimensions such as input column (X) and filter column (S) in CONV2D operations. Overlaid dimensions refer to reference and sliding dimensions within a sliding window.

A.1.2 Component-wise Flexibility

Definition 7. Available Iteration Orders

Given a layer size, L_{sz}, and the number of on-chip cluster levels, N_C, then iteration order choices, $\omega(L_{sz}, N_C)$, is as follows:

$$\omega(L_{sz}, N_C) = (len(L_{sz})!)^{N_C}$$

Definition 7 defines the number of possible iteration orders across N_C cluster levels on a chip. At each level, we consider permutation of all the independent data dimensions the layer has. For example, in CONV2D without batches, six dimensions exist so 6! choices at each level exist. The iteration order choice at each lavel is independent, so we compute the power of choices at each level.

Definition 8. Parallel Dimension Choices

Given a layer size, L_{sz}, and the number of on-chip cluster levels (N_C) the number of parallel dimension choices (π) is defined as follows:

$$\pi(L_{sz}, N_C) = \prod_{0 \leq C_{lv} < N_C} \binom{Len(L_{Sz}) - C_{lv}}{1}$$

In this definition, we exclude redundant cases that spatially partition a data dimension multiple times at multiple cluster levels since such clustering is equivalent to a flattened mapping. For example, if we have a 4x4 PE array, and tries to parallelize output channel across PE rows and columns (twice), the resulting mapping is equivalent to parallelizing output channel across a 1D PE array with 4x4=16 PEs. They result in different performance and cost, but from the logical mapping's perspective, they are equivalent. We does not count them separately when we estimate the flexibility.

Now we define the number of tiling choices.

Definition 9. Tiling Choices at Cluster Level C_{lv}

At cluster level C_{lv}, given the tile size of $\mathrm{T}(C_{lv})$, the number of tile size choices for level $C_{lv} - 1$, $\tau(\mathrm{T}(C_{lv}), C_{lv} - 1)$, is as follows:

$$\tau(\mathrm{T}(C_{lv}), C_{lv} - 1) = \prod_{\forall (d,sz) \in \mathrm{T}(C_{lv})} sz$$

Note that the number of tiling choices depend on the tile size at the upper level. That is, the lower level cannot have tile size larger than that of the upper level. Therefore, we first define the relationship between two adjacent tile sizes in Definition 9. Based on Definition 9, we define the entire number of tiling choices in Definition 10.

Definition 10. Tiling Choices

The number of tiling choices for a given layer size L_{sz} and number of on-chip cluster levels N_C, the number of tiling choices, $\tau_{all}(L_{sz}, N_C)$, is as follows:

$$\tau_{all}(L_{sz}, N_C) = \tau_{rec}(T_{sz} = L_{sz}, C_{lv} = N_C) =$$

$$\sum_{\forall \mathrm{T}(C_{lv})} \tau_{rec}(\mathrm{T}(C_{lv}), C_{lv} - 1)$$

where $\tau_{rec}(T_{sz}, 0) = 1$

Definition 9 indicates that the number of available tile size at a lower level cluster is constrained by the tile size at the upper level cluster. For example, if a mapping assigns indices of a dimension α in the range of [0, $\mathrm{T}(C_{Lv})[\alpha]$), one level below cluster cannot have tile size larger than $\mathrm{T}(C_{Lv})[\alpha]$ (i.e., $\mathrm{T}(C_{Lv} - 1)[\alpha] \leq \mathrm{T}(C_{Lv})[\alpha]$) because of out-of-range indices. Therefore, the number of choices depend on the tile sizes on the upper level cluster, which results in a recursive definition as shown in Definition 10. Also, note that the definition covers entire tile sizes including those do not divide the upper level tile sizes. For example, if $\mathrm{T}(3)[\alpha] = 64$, $\mathrm{T}(2)[\alpha]$ can be any number between 1 and 64.

Now we define component-wise flexibility based on the definitions we discussed in this subsection.

A.1.3 Mapping Flexibility

Based on base definitions we discussed in Section A.1.1, we define mapping flexibility.

Definition 11. Component-wise Flexibility

$$Flex_{order} = \frac{\omega_{supported}(Acc, L_{sz}, N_C)}{\omega(L_{sz}, N_C)}$$

$$Flex_{parallel} = \frac{\pi_{supported}(Acc, L_{sz}, N_C)}{\pi(L_{sz}, N_C)}$$

$$Flex_{tiling} = \frac{\tau_{all-supported}(Acc, L_{sz}, N_C)}{\tau_{all}(L_{sz}, N_C)}$$

where $\omega_{supported}(Acc, L_{sz}, N_C)$, $\pi_{supported}(Acc, L_{sz}, N_C)$, and $\tau_{all-supported}(Acc, L_{sz}, N_C)$ refer to the number of available iteration order, parallel dimension, and tile size choices by an accelerator Acc.

$Flex_{order}$, $Flex_{parallel}$, and $Flex_{tiling}$ refer to iteration order, parallel dimension, and tiling flexibility, respectively.

Definition 11 defines flexibility for each component of a mapping. The denominator of each refers to algorithmic choices, which are determined by the layer size and the high-level architecture template of a target accelerator (the number of on-chip cluster levels; N_C). High-level architecture template refers to the organization of an accelerator, which includes the number of memory hierarchy levels and PE dimensionality without exact design parameters for them (e.g., memory size, number of PE rows/columns, etc.). The numerators ($\omega_{supported}, \pi_{supported}$, and $\tau_{all-supported}$) of each refers to actually available choices constrained by a concrete accelerator (Acc) with all the design parameters.

Based on the component-wise flexibility definition, we define the mapping flexibility of an accelerator as the cross-product of all the component-wise flexibility.

Definition 12. Mapping Flexibility

For a given layer size (L_{Sz}) and the number of on-chip cluster levels (N_C) given by a high-level architecture template,

$$Flex_{mapping} = Flex_{order} \times Flex_{parallel} \times Flex_{tiling}$$

Since we have defined ω, π, and τ_{all}, in this subsection, we need $\omega_{supported}, \pi_{supported}$,

Hardware		Implication to Mapping								Costs and Benefits		
Hardware Type	Hardware Choice	Mapping Size						Spatial Dimension	Iteration Order	Performance Benefits	Performance Disadvantages	Hardware Cost
		K	C	Y	X	R	S					
Buffer	FIFO	-	-	≤Sz(R)	≤Sz(S)	-	-	-	-	-	Sequential Access	Control: O(1)
	Scratchpad	-	-	-	-	-	-	-	-	Random Access	-	Control: O(n)
Distribution NoC	Bus	-	-	-	-	-	-	C Unpreferred	SpDim: inner-most position not preferred	Multicast Support	Low BW	O(n)
	Crossbar	-	-	-	-	-	-	-	-	Full Bandwidth	-	$O(n^2)$
	Mesh	-	-	-	-	-	-	-	-	-	- Link Conflicts - Hop-by-hop traversal	$O(n^2)$
	Tree	-	-	-	-	-	-	C Unpreferred	SpDim: inner-most position not preferred	Multicast Support	-	O(n log(n))
	Store-and-Forward	-	-	-	-	-	-	-	-	Multicast Support	High Latency	O(n)
Reduction NoC	Temporal Reduction	-	-	-	-	-	-	C/R/S Prohibited	-	-	Low Reduction Bandwidth	O(1)
	Adder Tree	-	-	-	-	-	-	C/R/S Preferred	-	- High BW - Low Latency	-	O(n log(n))
	Reduce-and-Forward	1	-	≤Sz(R)	≤Sz(S)	-	-	C/R/S Preferred	-	Pipelined Reduction	High Latency	O(n)

Figure A.2: Hardware choices and their implication to the mapping flexibility. "-" refers that no implication or no particular benefits/disadvantages.

and $\tau_{all-supported}$ for computing the mapping flexibility. We discuss how hardware choices impact them next.

A.2 Hardware Constraints on Available Mappings

In the previous sections, Section A.1, we discussed how we can quantify the flexibility. The definition of flexibility for each component of a mapping, Definition 11, implied that the number of possible choices can be constrained by hardware choices. In this section, we discuss how hardware choices affect the available choices for each mapping component. Since we focus on data orchestration, we do not explore compute unit choices, but focus on buffer, distribution NoC, and reduction NoC choices.

Figure A.2 summarizes the impact of hardware choices on mapping flexibility and cost/benefit of an accelerator.

A.2.1 Buffer Choices

For buffers, we analyze two dominant buffer types in DNN accelerators: FIFO and scratch-pad. Regardless of buffer types, buffer size constraint the mapping size. We omit the size

constraint in Figure A.2 but state the memory size constraint as follows:

Theorem 1. Memory Size Constraint

Given a tile size $\mathrm{T}(C_{lv})$ at cluster level C_{lv}, the following inequalities must hold:

$$\forall C_{lv} > 1,$$

$$\mathrm{T}(C_{lv})[K] \times \mathrm{T}(C_{lv})[C] \times \mathrm{T}(C_{lv})[Y] \times \mathrm{T}(C_{lv})[X] \leq 0.5 \times MemSz_{Filter}(C_{lv})$$

$$\mathrm{T}(C_{lv})[C] \times \mathrm{T}(C_{lv})[Y] \times \mathrm{T}(C_{lv})[X] \leq 0.5 \times MemSz_{Input}(C_{lv})$$

$$\mathrm{T}(C_{lv})[K] \times \mathrm{T}(C_{lv})[Y'] \times \mathrm{T}(C_{lv})[X'] \leq 0.5 \times MemSz_{Output}(C_{lv})$$

where $MemSz_t(C_{lv})$ refers to the memory size for tensor t at a cluster level C_{lv}, and $\mathrm{T}(C_{lv})[\alpha]$ refers to the tile size of dimension α at the cluster level C_{lv}.

Theorem 1 assumes that we have hard-partitioned buffers for each tensor. Theorem 1 states the constraint on CONV2D operations. In general, Theorem 1 states that the product of tile sizes of coupled dimensions of each tensor, which is total number of mapped data points of a tensor, needs to be smaller than the memory size for the tensor. Theorem 1 multiplies 0.5 to the memory size for double buffering. Taking Theorem 1 as an underlying condition, we analyze extra constraints by the choice of buffers.

FIFO is often used at the lowest hierarchy of the memory, before ALUs, for their low costs. However, FIFO does not allow random access and force sequential accesses of data. We target non-circular FIFO since circular FIFOs require high latency for random accesses, which often makes a circular FIFO an undesirable option. To prevent re-circulation of data within FIFO, FIFOs have constraints on mapping sizes. First, if multiple sliding windows are processed within a data tile, we need to circulate over filter values. To prohibit it, mapping size on input row (Y) and input column (X) dimensions are constrained to be less than or equal to the sliding window (or, receptive field) sizes, filter row (R) and filter column (S),

respectively. Note that such constraints apply for the overlaid dimensions (Y-R and X-S dimensions in CONV2D) since different combinations of data indices on those dimensions can point to the same data, as the definition in Definition 6 shows.

Scratchpad, in contrast, provides random access capability at the cost of control logic and wires. Although scratchpad is costlier than FIFO, scratchpad is a flexible buffer that does not impose any constraint to the available mapping.

A.2.2 Distribution NoC Choices

For distribution NoCs, which delivers data from upstream buffer (DRAM side) to downstream buffer (PE side) at each hierarchy level, we analyze bus, crossbar, mesh, tree, and store-and-forward (systolic array style).

One of the major differences of those choices is the multicast capability. Although none of them imposes constraint to mappings, distribution NoCs with multicast support prefers specific parallel dimensions and iteration order, so that they can exploit their multicast mechanism. For example, if input channel (C) dimension is spatially mapped (i.e., parallelized) at the inner-most loop position, multicast capability is never utilized. This is because the input channel (C) dimension is coupled with both input activation and filter, so different data points of input activation and filter are required for each sub-cluster (or, PE at the lowest level).

The impact of Distribution NoC choices is mainly on the performance and hardware cost. We summarize those in costs and benefits column in Figure A.2, based on the observation in Chapter 4.

A.2.3 Reduction NoC Choices

We consider three choices; temporal reduction, adder tree, and reduce-and-forward (systolic array style). Unlike distribution NoCs, the choice of reduction NoC imposes constraints on spatial dimensions. For example, if input channel dimension (C) is spatially mapped

(parallelized), partial outputs for the same output activation data point will be generated by multiple sub-clusters (or, PEs at the lowest cluster level). Since they need to be accumulated, reduction across sub-clusters (or PEs) is required, which implies the need for spatial reduction capability. Adder tree and reduce-and-forward provide spatial reduction capability while temporal reduction does not. Therefore, parallelizing data dimensions to be accumulated to generate final output activation values (e.g., C/R/S dimensions in CONV2D) is prohibited for temporal reduction. However, parallelizing such dimensions is preferred by adder tree and reduce-and-forward to exploit their spatial reduction capabilities. Also, note that such preference indicates a direction to local optimal point in the design space, which may or may not lead to a globally optimal design point.

A.3 Summary and Future Works

In Section A.1, we defined a metric to measure mapping flexibility of DNN accelerators. We also analyzed how hardware choices constraint the available mappings in Section A.2. We observe that each hardware component for distribution/reduction NoC and buffers either constraint available mappings or prefer specific features of mappings.

Although each component has such constraints or preference toward mappings, they need to be considered in whole to analyze the performance and energy efficiency of a mapping on a DNN accelerator. Therefore, we plan to develop a framework that can quantify the benefits of flexible mappings enabled by various hardware components and their cost. Using the framework, we will explore the costs and benefits of the flexibility in DNN accelerators, which will reveal the desired amount of flexibility for various DNN workloads.

REFERENCES

[1] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[3] C. Szegedy *et al.*, "Going deeper with convolutions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *arXiv preprint arXiv:1801.04381*, 2019.

[6] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated Residual Transformations for Deep Neural Networks," *arXiv preprint arXiv:1611.05431*, 2017.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[8] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[9] A. Toshev and C. Szegedy, "Deeppose: Human pose estimation via deep neural networks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[10] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *PAMI*, vol. 35, no. 8, pp. 1915–1929, 2013.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," *arXiv preprint arXiv:1512.02595*, 2015.

[13] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[15] L. M. Alec Radford and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv preprint arXiv:1511.06434*, 2015.

[16] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 1–12.

[18] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Madison,WI, Mar. 2019.

[19] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2019, pp. 754–768.

[20] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.

[21] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[22] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 461–475.

[23] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, IEEE Press, 2018, pp. 650–661.

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, 2018, pp. 578–594, ISBN: 978-1-931971-47-8.

[25] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, ACM, 2017, pp. 45–54.

[26] D. Patterson, *Domain-specific architectures for deep neural networks*, https://inst.eecs.berkeley.edu//~cs152/sp19/lectures/L20-DSA.pdf, 2019.

[27] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[28] H. Kwon and T. Krishna, "Opensmart: Single-cycle multi-hop noc generator in bsv and chisel," 2017.

[29] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *International Symposium on Networks-on-Chip (NOCS)*, 2017.

[30] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "Mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2019, pp. 282–292.

[31] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *2017 Eleventh IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, IEEE, 2017, pp. 1–8.

[32] H. Kwon, L. Lai, T. Krishna, and V. Chandra, "Herald: Optimizing heterogeneous dnn accelerators for edge devices," *arXiv preprint arXiv:1909.07437*, 2019.

[33] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi, "Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks," in *Design Automation Conference (DAC)*, 2020.

[34] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 58–70.

[35] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A Data-centric Compiler for DNN Operators on Spatial Accelerators," *arXiv preprint arXiv:2002.07752*, 2020.

[36] F. Muñoz-Martínez, L. J. Abellán, E. M. Acacio, and T. Krishna, "STONNE: A Detailed Architectural Simulator for Flexible Neural Network Accelerators," *arXiv preprint arXiv:2006.07137*, 2020.

[37] R. Guirado, H. Kwon, E. Alarcón, S. Abadal, and T. Krishna, "Understanding the impact of on-chip communication on dnn accelerator performance," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, IEEE, 2019, pp. 85–88.

[38] *Maeri: Enabling rapid design space exploration and prototyping of dnn acceleratorsip*, http://synergy.ece.gatech.edu/tools/maeri/maeri_tutorial_isca2018/, 2018.

[39] *Enabling rapid design space exploration and prototyping of dnn acceleratorsfazer blog*, http://synergy.ece.gatech.edu/tools/maeri/maeri-tutorial-hpca-2019/, 2019.

[40] *Enabling rapid design space exploration and prototyping of dnn acceleratorsfazer blog*, http://synergy.ece.gatech.edu/tools/maeri/maeri-tutorial-hpca-2019/, 2019.

[41] T. Krishna, H. Kwon, M. Pellauer, A. Parashar, and A. Samajdar, *Data Orchestration inDeep Learning Accelerators*. Morgan & Claypool Publishers, To be published in 2020.

[42] *The future is here - iphone x (neural engine)*, https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/, 2017.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[44] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners,"

[45] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[46] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[47] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[48] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, IEEE, 2020, pp. 488–501.

[49] *Nvdla deep learning accelerator*, http://nvdla.org, 2017.

[50] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Design Automation Conference (DAC)*, 2016, pp. 1–6.

[51] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[52] I. Kodukula and K. Pingali, "Data-centric transformations for locality enhancement," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 319–364, Jun. 2001.

[53] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric Multi-level Blocking," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97, Las Vegas, Nevada, USA: ACM, 1997, pp. 346–357, ISBN: 0-89791-907-6.

[54] I. Kodukula, K. Pingali, R. Cox, and D. Maydan, "An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99, Rhodes, Greece: ACM, 1999, pp. 482–491, ISBN: 1-58113-164-X.

[55] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[56] D. Vainbrand and R. Ginosar, "Network-on-chip architectures for neural networks," in *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, IEEE, 2010, pp. 135–144.

[57] J. Harkin, F. Morgan, S. Hall, P. Dudek, T. Dowrick, and L. McDaid, "Reconfigurable platforms and the challenges for large-scale implementations of spiking neural networks," in *2008 International Conference on Field Programmable Logic and Applications*, IEEE, 2008, pp. 483–486.

[58] T. Theocharides, G. Link, N. Vijaykrishnan, M. Invin, and V. Srikantam, "A generic reconfigurable neural network architecture as a network on chip," in *IEEE International SOC Conference, 2004. Proceedings.*, IEEE, 2004, pp. 191–194.

[59] R. Emery, A. Yakovlev, and G. Chester, "Connection-centric network for spiking neural networks," in *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, IEEE, 2009, pp. 144–152.

[60] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *International conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2014.

[61] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[62] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[63] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[64] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *ISCA*, 2013, pp. 24–35.

[65] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015, pp. 161–170.

[66] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016, pp. 1–13.

[67] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA*, 2014, pp. 97–108.

[68] L. Song *et al.*, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *DAC*, 2016.

[69] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[70] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *44th International Symposium on Computer Architecture (ISCA)*, 2017.

[71] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[72] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 161–170.

[73] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016, pp. 1–13.

[74] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[75] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[76] M. I. Jordan, "Serial order: A parallel distributed processing approach," *Advances in psychology*, vol. 121, pp. 471–495, 1997.

[77] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[78] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *IEEE Neural Networks*, vol. 1, 1996, pp. 347–352.

[79] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *ASP-DAC*, 2017, pp. 629–634.

[80] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *FCCM*, 2015, pp. 111–118.

[81] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "Fpga-based low-power speech recognition with recurrent neural networks," in *SiPS*, 2016, pp. 230–235.

[82] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *FPGA*, 2017, pp. 75–84.

[83] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[84] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation," *IEEE Transactions on Neural Networks*, vol. 16, no. 6, pp. 1664–1672, 2005.

[85] R. Tavcar, J. Dedic, D. Bokal, and A. Zemva, "Transforming the lstm training algorithm for efficient fpga-based adaptive control of nonlinear dynamic systems," *Informacije Midem-Journal of Microelectronics Electronic Components and Materials*, vol. 43, no. 2, pp. 131–138, 2013.

[86] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware," in *ISLPED*, 2016, pp. 168–173.

[87] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *ISSCC*, 2017, pp. 240–241.

[88] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks (ICANN)*, Springer, 2014, pp. 281–290.

[89] V. Sarkar, "Automatic selection of high-order transformations in the IBM XL FORTRAN compilers," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 233–264, 1997.

[90] V. Sarkar and N. Megiddo, "An analytical model for loop tiling and its solution," in *ISPASS*, 2000, pp. 146–153.

[91] J. Shirako, K. Sharma, N. Fauzia, L. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical Bounds for Optimal Tile Size Selection," in *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, 2012, pp. 101–121.

[92] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91, Toronto, Ontario, Canada: ACM, 1991, pp. 30–44, ISBN: 0-89791-428-7.

[93] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 101–113, ISBN: 978-1-59593-860-2.

[94] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, 2010, pp. 1–11.

[95] J. Shirako, L.-N. Pouchet, and V. Sarkar, "Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, New Orleans, Louisana: IEEE Press, 2014, pp. 287–298, ISBN: 978-1-4799-5500-8.

[96] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, and P. Sadayappan, "Analytical Modeling of Cache Behavior for Affine Programs," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 32:1–32:26, Dec. 2017.

[97] *Maestro project page*, https://synergy.ece.gatech.edu/tools/maestro/, 2019.

[98] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, pp. 22–31, 2009.

[99] Wu, Yannan N. and Emer, Joel S. and Sze, Vivienne, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2019.

[100] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, Springer, 2015, pp. 234–241.

[101] F. Akopyan *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *TCAD*, 2015.

[102] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *ISPD*, 2015.

[103] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 369–381.

[104] T. Krishna, C.-H. O. Chen, W. C. Kwon, and L.-S. Peh, "Breaking the on-chip latency barrier using smart," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 378–389.

[105] ——, "Breaking the on-chip latency barrier using smart," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 378–389.

[106] R. Nikhil, "Bluespec system verilog: Efficient, correct rtl from high level specifications," in *International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2004.

[107] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[108] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, J. Wanderer, U. Holzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 183–197, 2015.

[109] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 0–0.

[110] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 331–344.

[111] S. Lee, S. W. Oh, D. Won, and S. J. Kim, "Copy-and-paste networks for deep video inpainting," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 4413–4421.

[112] V. Ramanishka, Y.-T. Chen, T. Misu, and K. Saenko, "Toward driving scene understanding: A dataset for learning driver behavior and causal reasoning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7699–7707.

[113] L. He, G. Wang, and Z. Hu, "Learning depth from single images with deep neural network embedding focal length," *IEEE Transactions on Image Processing*, vol. 27, no. 9, pp. 4676–4689, 2018.

[114] M. Madadi, S. Escalera, X. Baró, and J. Gonzalez, "End-to-end global to local cnn learning for hand pose recovery in depth data," *arXiv preprint arXiv:1705.09606*, 2017.

[115] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[116] Qualcomm, *Quacomm hexagon 680*, https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.24-Monday-Epub/HC27.24.20-Multimedia-Epub/HC27.24.211-Hexagon680-Codrescu-Qualcomm.pda, 2015.

[117] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 535–547.

[118] K. Olukotun, *Designing computer systems for software 2.0*, https://iscaconf.org/isca2018/docs/Kunle-ISCA-Keynote-2018.pdf, 2018.

[119] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Ventakesan, K. S. W., C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *International conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2019.

[120] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher, "Morph: Flexible acceleration for 3d cnn-based video understanding," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 933–946.

VITA

Hyoukjun Kwon was born on December 22, 1988 in Seoul, Republic of Korea. He received his Bachelor's Degrees in Environmental Material Science and Computer Science and Engineering in 2015 from Seoul National University. He received his PhD from the School of Computer Science at Georgia Institute of Technology. He was advised by Professor Tushar Krishna, the principal investigator of Synergy Lab where Hyoukjun was a member.

Hyoukjun's research interests are in the intersection of computer architecture, compiler, and machine learning. In particular, Hyoukjun is interested in co-design of all of three for specialized computing stacks that involve accelerators. Applications in his interests include not only machine learning, but also for high-performance computing applications such as graph, tensor contraction, and so on. Several open source projects based on Hyoukjun's research were presented in tutorials at ISCA 2018 and HPCA 2019, and led to attentions from both academia and industry. His thesis works also led to a book publication, which is scheduled to be published in 2020. One of Hyoukjun's paper received honorable mention from Top Picks in Computer Architecture Conferences in 2018, and another was selected as Top Picks in Compute Architecture Conferences in 2019.. Hyoukjun also won the best paper award at HPCA 2020.